eBook Gratuit

APPRENEZ

Elixir Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#elixir

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: elixir-language

It is an unofficial and free Elixir Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Elixir Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Premiers pas avec Elixir Language

Remarques

Elixir est un langage dynamique et fonctionnel conçu pour créer des applications évolutives et maintenables.

Elixir exploite la machine virtuelle Erlang, connue pour exécuter des systèmes à faible latence, distribués et tolérants aux pannes, tout en étant utilisée avec succès dans le développement Web et le domaine des logiciels intégrés.

Versions

Version	Date de sortie
0,9	2013-05-23
1.0	2014-09-18
1.1	2015-09-28
1.2	2016-01-03
1.3	2016-06-21
1.4	2017-01-05

Examples

Bonjour le monde

Pour les instructions d'installation sur elixir, cliquez ici , il décrit les instructions relatives aux différentes plates-formes.

Elixir est un langage de programmation créé avec `erlang` , et utilise le temps d'exécution `BEAM` d'erlang (comme `JVM` pour java).

Nous pouvons utiliser elixir dans deux modes: shell interactif `iex` ou exécution directe à l'aide de la commande `elixir` .

Placez les éléments suivants dans un fichier nommé `hello.exs` :

```
IO.puts "Hello world!"
```

À partir de la ligne de commande, tapez la commande suivante pour exécuter le fichier source Elixir:

```
$ elixir hello.exs
```

Cela devrait sortir:

> Bonjour le monde!

Ceci est connu comme le *mode scripté* d' `Elixir` . En fait, les programmes Elixir peuvent également être compilés (et généralement, ils le sont) en bytecode pour la machine virtuelle BEAM.

Vous pouvez également utiliser `iex` pour un shell interactif d'élixir (recommandé), exécutez la commande pour obtenir une invite comme celle-ci:

```
Interactive Elixir (1.3.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Ici vous pouvez essayer vos exemples de `hello world` elixir:

```
iex(1)> IO.puts "hello, world"
hello, world
:ok
iex(2)>
```

Vous pouvez aussi compiler et exécuter vos modules via `iex` . Par exemple, si vous avez un `helloworld.ex` qui contient:

```
defmodule Hello do
   def sample do
       IO.puts "Hello World!"
   end
end
```

À travers `iex` , faites:

```
iex(1)> c("helloworld.ex")
[Hello]
iex(2)> Hello.sample
Hello World!
```

Bonjour tout le monde de IEx

Vous pouvez également utiliser le `IEx` (Interactive Elixir) pour évaluer des expressions et exécuter du code.

Si vous êtes sous Linux ou Mac, tapez simplement iex sur votre compte et appuyez sur Entrée:

```
$ iex
```

Si vous êtes sur une machine Windows, tapez:

```
C:\ iex.bat
```

Ensuite, vous entrerez dans la REPL IEx (lecture, évaluation, impression, boucle), et vous pouvez simplement taper quelque chose comme:

```
iex(1)> "Hello World"
"Hello World"
```

Si vous souhaitez charger un script lors de l'ouverture d'une REPL IEx, vous pouvez le faire:

```
$ iex script.exs
```

Étant donné que `script.exs` est votre script. Vous pouvez maintenant appeler des fonctions à partir du script dans la console.

Lire Premiers pas avec Elixir Language en ligne: https://riptutorial.com/fr/elixir/topic/954/premiers-pas-avec-elixir-language

Chapitre 2: Cartes et listes de mots clés

Syntaxe

- map =% {} // crée une carte vide
- map =% {: a => 1,: b => 2} // crée une carte non vide
- list = [] // crée une liste vide
- list = [{: a, 1}, {: b, 2}] // crée une liste de mots-clés non vide

Remarques

Elixir fournit deux structures de données associatives: les *cartes* et les *listes de mots clés* .

Les cartes sont du type clé-valeur Elixir (également appelé dictionnaire ou hachage dans d'autres langues).

Les listes de mots-clés sont des tuples de clé / valeur qui associent une valeur à une clé donnée. Ils sont généralement utilisés comme options pour un appel de fonction.

Examples

Créer une carte

Les cartes sont du type clé-valeur Elixir (également appelé dictionnaire ou hachage dans d'autres langues). Vous créez une carte en utilisant la syntaxe `%w{}` :

```
%{} // creates an empty map
%{:a => 1, :b => 2} // creates a non-empty map
```

Les clés et les valeurs peuvent utiliser n'importe quel type:

```
%{"a" => 1, "b" => 2}
%{1 => "a", 2 => "b"}
```

De plus, vous pouvez avoir des cartes avec des types mixtes pour les clés et les valeurs ":

```
// keys are integer or strings
%{1 => "a", "b" => :foo}
// values are string or nil
%{1 => "a", 2 => nil}
```

Lorsque toutes les clés d'une carte sont des atomes, vous pouvez utiliser la syntaxe du mot clé pour plus de commodité:

```
%{a: 1, b: 2}
```

Création d'une liste de mots clés

Les listes de mots-clés sont des tuples de clé / valeur, généralement utilisés comme options pour un appel de fonction.

```
[{:a, 1}, {:b, 2}] // creates a non-empty keyword list
```

La même clé peut être répétée plusieurs fois.

```
[{:a, 1}, {:a, 2}, {:b, 2}]
[{:a, 1}, {:b, 2}, {:a, 2}]
```

Les clés et les valeurs peuvent être de tout type:

```
[{"a", 1}, {:a, 2}, {2, "b"}]
```

Différence entre les cartes et les listes de mots clés

Les cartes et les listes de mots-clés ont une application différente. Par exemple, une carte ne peut pas avoir deux clés de même valeur et elle n'est pas ordonnée. Inversement, une liste de mots-clés peut être un peu difficile à utiliser dans certains cas.

Voici quelques cas d'utilisation pour les cartes et les listes de mots clés.

Utilisez les listes de mots-clés lorsque:

- vous avez besoin des éléments à commander
- vous avez besoin de plus d'un élément avec la même clé

Utilisez des cartes lorsque:

- vous voulez faire correspondre les modèles à certaines clés / valeurs
- vous n'avez pas besoin de plus d'un élément avec la même clé
- chaque fois que vous n'avez pas explicitement besoin d'une liste de mots clés

Lire Cartes et listes de mots clés en ligne: https://riptutorial.com/fr/elixir/topic/2706/cartes-et-listes-de-mots-cles

Chapitre 3: Comportements

Examples

introduction

Les comportements sont une liste de spécifications de fonctions qu'un autre module peut implémenter. Ils sont similaires aux interfaces dans d'autres langues.

Voici un exemple de comportement:

```
defmodule Parser do
  @callback parse(String.t) :: any
  @callback extensions() :: [String.t]
end
```

Et un module qui l'implémente:

```
defmodule JSONParser do
  @behaviour Parser

  def parse(str), do: # ... parse JSON
  def extensions, do: ["json"]
end
```

L' `@behaviour` module `@behaviour` ci-dessus indique que ce module est censé définir toutes les fonctions définies dans le module Analyseur. Les fonctions manquantes entraîneront des erreurs de compilation de fonctions de comportement non définies.

Les modules peuvent avoir plusieurs attributs `@behaviour` .

Lire Comportements en ligne: https://riptutorial.com/fr/elixir/topic/3558/comportements

Chapitre 4: Conditionnels

Remarques

Notez que la syntaxe `do...end` est le sucre syntaxique pour les listes de mots clés classiques, vous pouvez donc le faire:

```
unless false, do: IO.puts("Condition is false")
# Outputs "Condition is false"

# With an `else`:
if false, do: IO.puts("Condition is true"), else: IO.puts("Condition is false")
# Outputs "Condition is false"
```

Examples

Cas

```
 case {1, 2} do
  {3, 4} ->
    "This clause won't match."
  {1, x} ->
    "This clause will match and bind x to 2 in this clause."
  _ ->
    "This clause would match any value."
end
```

`case` est uniquement utilisé pour correspondre au modèle donné des données particulières. Ici, `{1,2}` correspond à un modèle de cas différent donné dans l'exemple de code.

si et à moins

```
if true do
    "Will be seen since condition is true."
end

if false do
    "Won't be seen since condition is false."
else
    "Will be seen.
end

unless false do
    "Will be seen."
end

unless true do
    "Won't be seen."
else
    "Will be seen."
end
```

cond

```
cond do
   0 == 1 -> IO.puts "0 = 1"
   2 == 1 + 1 -> IO.puts "1 + 1 = 2"
   3 == 1 + 2 -> IO.puts "1 + 2 = 3"
end

# Outputs "1 + 1 = 2" (first condition evaluating to true)
```

`cond` va `CondClauseError` une `CondClauseError` si aucune condition n'est vraie.

```
cond do
  1 == 2 -> "Hmmm"
  "foo" == "bar" -> "What?"
end
# Error
```

Cela peut être évité en ajoutant une condition qui sera toujours vraie.

```
cond do
  ... other conditions
  true -> "Default value"
end
```

À moins que l'on ne s'attende jamais à ce qu'il atteigne le cas par défaut, le programme devrait en fait tomber en panne à ce stade.

avec clause

`with` clause `with` est utilisée pour combiner des clauses de correspondance. Il semble que nous combinions des fonctions anonymes ou des fonctions de gestion avec plusieurs corps (clauses correspondantes). Considérez le cas: nous créons un utilisateur, l'insérons dans la base de données, puis créons un message électronique d'accueil et l'envoyons à l'utilisateur.

Sans la clause `with` , nous pourrions écrire quelque chose comme ceci (j'ai omis les implémentations de fonctions):

```
case create_user(user_params) do
  {:ok, user} ->
    case Mailer.compose_email(user) do
      {:ok, email} ->
        Mailer.send_email(email)
      {:error, reason} ->
        handle_error
    end
  {:error, changeset} ->
    handle_error
end
```

Ici, nous traitons le flux de notre processus métier avec la `case` (il pourrait être `cond` ou `if`). Cela nous conduit à ce qu'on appelle la «pyramide de malheur» , car nous devons faire face aux

conditions possibles et décider: aller plus loin ou non. Il serait beaucoup plus intéressant de réécrire ce code avec `with` :

```
with {:ok, user} <- create_user(user_params),
     {:ok, email} <- Mailer.compose_email(user) do
  {:ok, Mailer.send_email}
else
  {:error, _reason} ->
    handle_error
end
```

Dans l'extrait de code ci-dessus, nous avons réécrit les clauses de `case` imbriquées avec `with` . Au sein `with` nous invoquons certaines fonctions (anonymes ou nommés) et correspondance de motif sur leurs sorties. Si tout adapté, `with` le retour `do` fait bloc, ou `else` résultat bloc autrement.

Nous pouvons omettre d' `else` façon `with` renverra soit `do` résultat de bloc ou le premier échec résultat.

Ainsi, la valeur de `with` instruction est le résultat de son bloc `do` .

Lire Conditionnels en ligne: https://riptutorial.com/fr/elixir/topic/2118/conditionnels

Chapitre 5: Conseils de débogage

Examples

Déboguer avec IEX.pry / 0

`IEx.pry/0` avec `IEx.pry/0` est assez simple.

1. `require IEx` dans votre module
2. Trouvez la ligne de code que vous voulez inspecter
3. Ajouter `IEx.pry` après la ligne

Maintenant, lancez votre projet (par exemple, le `iex -S mix`).

Lorsque la ligne avec `IEx.pry/0` est atteinte, le programme s'arrête et vous avez la possibilité d'inspecter. C'est comme un point d'arrêt dans un débogueur traditionnel.

Lorsque vous avez terminé, tapez simplement `respawn` dans la console.

```
require IEx;

defmodule Example do
  def double_sum(x, y) do
    IEx.pry
    hard_work(x, y)
  end

  defp hard_work(x, y) do
    2 * (x + y)
  end
end
```

Déboguer avec IO.inspect / 1

Il est possible d'utiliser IO.inspect / 1 comme outil pour déboguer un programme d'élixir.

```
defmodule MyModule do
  def myfunction(argument_1, argument_2) do
    IO.inspect(argument_1)
    IO.inspect(argument_2)
  end
end
```

Il imprimera argument_1 et argument_2 sur la console. Comme `IO.inspect/1` renvoie son argument, il est très facile de l'inclure dans les appels de fonctions ou les pipelines sans casser le flux:

```
do_something(a, b)
|> do_something_else(c)
```

```
# can be adorned with IO.inspect, with no change in functionality:

do_something(IO.inspect(a), IO.inspect(b))
|> IO.inspect
do_something(IO.inspect(c))
```

Debug dans pipe

```
defmodule Demo do
  def foo do
    1..10
    |> Enum.map(&(&1 * &1))           |> p
    |> Enum.filter(&rem(&1, 2) == 0) |> p
    |> Enum.take(3)                   |> p
  end

  defp p(e) do
    require Logger
    Logger.debug inspect e, limit: :infinity
    e
  end
end
```

```
iex(1)> Demo.foo

23:23:55.171 [debug] [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

23:23:55.171 [debug] [4, 16, 36, 64, 100]

23:23:55.171 [debug] [4, 16, 36]

[4, 16, 36]
```

Pry en pipe

```
defmodule Demo do
  def foo do
    1..10
    |> Enum.map(&(&1 * &1))
    |> Enum.filter(&rem(&1, 2) == 0) |> pry
    |> Enum.take(3)
  end

  defp pry(e) do
    require IEx
    IEx.pry
    e
  end
end
```

```
iex(1)> Demo.foo
Request to pry #PID<0.117.0> at lib/demo.ex:11

      def pry(e) do
```

```
        require IEx
        IEx.pry
        e
      end

Allow? [Yn] Y

Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
pry(1)> e
[4, 16, 36, 64, 100]
pry(2)> respawn

Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
[4, 16, 36]
iex(1)>
```

Lire Conseils de débogage en ligne: https://riptutorial.com/fr/elixir/topic/2719/conseils-de-debogage

Chapitre 6: Conseils et astuces pour la console IEx

Examples

Recompiler le projet avec `recompile`

```
iex(1)> recompile
Compiling 1 file (.ex)
:ok
```

Voir la documentation avec `h`

```
iex(1)> h List.last

                                def last(list)

Returns the last element in list or nil if list is empty.

Examples

| iex> List.last([])
| nil
|
| iex> List.last([1])
| 1
|
| iex> List.last([1, 2, 3])
| 3
```

Récupère la valeur de la dernière commande avec `v`

```
iex(1)> 1 + 1
2
iex(2)> v
2
iex(3)> 1 + v
3
```

Voir aussi: Obtenir la valeur d'une ligne avec `v`

Récupère la valeur d'une commande précédente avec `v`

```
iex(1)> a = 10
10
iex(2)> b = 20
20
iex(3)> a + b
30
```

Vous pouvez obtenir une ligne spécifique en passant l'index de la ligne:

```
iex(4)> v(3)
30
```

Vous pouvez également spécifier un index relatif à la ligne en cours:

```
iex(5)> v(-1) # Retrieves value of row (5-1) -> 4
30
iex(6)> v(-5) # Retrieves value of row (5-4) -> 1
10
```

La valeur peut être réutilisée dans d'autres calculs:

```
iex(7)> v(2) * 4
80
```

Si vous spécifiez une ligne non existante, `IEx` une erreur:

```
iex(7)> v(100)
** (RuntimeError) v(100) is out of bounds
    (iex) lib/iex/history.ex:121: IEx.History.nth/2
    (iex) lib/iex/helpers.ex:357: IEx.Helpers.v/1
```

Quittez la console IEx

1. Utilisez Ctrl + C, Ctrl + C pour quitter

```
iex(1)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

2. Utilisez `Ctrl+ \` pour quitter immédiatement

Voir les informations avec `i`

```
iex(1)> i :ok
Term
  :ok
Data type
  Atom
Reference modules
  Atom
iex(2)> x = "mystring"
"mystring"
iex(3)> i x
Term
  "mystring"
Data type
  BitString
Byte size
  8
```

```
Description
  This is a string: a UTF-8 encoded binary. It's printed surrounded by
  "double quotes" because all UTF-8 encoded codepoints in it are printable.
Raw representation
  <<109, 121, 115, 116, 114, 105, 110, 103>>
Reference modules
  String, :binary
```

Création de PID

Ceci est utile lorsque vous n'avez pas stocké le PID d'une commande précédente

```
iex(1)> self()
#PID<0.138.0>
iex(2)> pid("0.138.0")
#PID<0.138.0>
iex(3)> pid(0, 138, 0)
#PID<0.138.0>
```

Ayez vos alias prêts lorsque vous démarrez IEx

Si vous placez vos alias couramment utilisés dans un fichier `.iex.exs` à la racine de votre application, IEx les chargera au démarrage.

```
alias App.{User, Repo}
```

Histoire persistante

Par défaut, l'historique des entrées utilisateur dans `IEx` ne persiste pas dans les différentes sessions.

`erlang-history` ajoute le support de l'historique à la fois au shell Erlang et à `IEx` :

```
git clone git@github.com:ferd/erlang-history.git
cd erlang-history
sudo make install
```

Vous pouvez maintenant accéder à vos entrées précédentes à l'aide des flèches haut et bas, même sur différentes sessions `IEx` .

Quand la console Elixir est bloquée ...

Parfois, vous risquez de lancer accidentellement quelque chose dans le shell qui finit par attendre pour toujours, bloquant ainsi le shell:

```
iex(2)> receive do _ -> :stuck end
```

Dans ce cas, appuyez sur Ctrl-g. Tu verras:

```
User switch command
```

Entrez ces commandes dans l'ordre:

- `k` (pour tuer le processus shell)
- `s` (pour démarrer un nouveau processus shell)
- `c` (pour se connecter au nouveau processus shell)

Vous allez vous retrouver dans une nouvelle coque Erlang:

```
Eshell V8.0.2  (abort with ^G)
1>
```

Pour démarrer un shell Elixir, tapez:

```
'Elixir.IEx.CLI':local_start().
```

(n'oubliez pas le dernier point!)

Ensuite, vous verrez apparaître un nouveau processus shell Elixir:

```
Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> "I'm back"
"I'm back"
iex(2)>
```

Pour échapper au mode "en attente de plus d'entrées" (dû à des guillemets non **`#iex:break`** , des crochets, etc.), tapez **`#iex:break`** , suivi d'un retour chariot (␍):

```
iex(1)> "Hello, "world"
...(1)>
...(1)> #iex:break
** (TokenMissingError) iex:1: incomplete expression

iex(1)>
```

Ce qui précède est particulièrement utile lorsque le copier-coller d'un fragment de code relativement volumineux transforme la console en mode «attente d'une entrée supplémentaire».

sortir de l'expression incomplète

Lorsque vous avez entré quelque chose dans IEx qui attend une complétion, telle qu'une chaîne multiligne, IEx changera l'invite pour indiquer qu'elle attend que vous finissiez en changeant l'invite pour avoir des points de suspension (`...`) plutôt que `iex` .

Si vous trouvez que IEx attend que vous finissiez une expression mais que vous ne savez pas exactement ce qu'il faut pour terminer l'expression ou que vous souhaitiez simplement abandonner cette ligne d'entrée, entrez `#iex:break` comme entrée de la console. Cela entraînera IEx à lancer une `TokenMissingError` et à annuler l'attente d'une entrée supplémentaire, vous ramenant ainsi à une entrée de console standard "de niveau supérieur".

```
iex:1> "foo"
"foo"
iex:2> "bar
...:2> #iex:break
** (TokenMissingError) iex:2: incomplete expression
```

Plus d'informations sont disponibles dans la documentation IEx .

Charger un module ou un script dans la session IEx

Si vous avez un fichier d'élixir; un script ou un module et que vous souhaitez le charger dans la session IEx en cours, vous pouvez utiliser la méthode `c/1` :

```
iex(1)> c "lib/utils.ex"
iex(2)> Utils.some_method
```

Cela compilera et chargera le module dans IEx, et vous pourrez appeler toutes ses méthodes publiques.

Pour les scripts, il exécutera immédiatement le contenu du script:

```
iex(3)> c "/path/to/my/script.exs"
Called from within the script!
```

Lire Conseils et astuces pour la console IEx en ligne:
https://riptutorial.com/fr/elixir/topic/1283/conseils-et-astuces-pour-la-console-iex

Chapitre 7: Cordes

Remarques

Une `String` dans Elixir est un binaire codé en `UTF-8` .

Examples

Convertir en chaîne

Utilisez Kernel.inspect pour convertir n'importe quoi en chaîne.

```
iex> Kernel.inspect(1)
"1"
iex> Kernel.inspect(4.2)
"4.2"
iex> Kernel.inspect %{pi: 3.14, name: "Yos"}
"%{pi: 3.14, name: \"Yos\"}"
```

Obtenir une sous-chaîne

```
iex> my_string = "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
iex> String.slice my_string, 6..10
"ipsum"
```

Diviser une chaîne

```
iex> String.split("Elixir, Antidote, Panacea", ",")
["Elixir", "Antidote", "Panacea"]
```

Interpolation de chaîne

```
iex(1)> name = "John"
"John"
iex(2)> greeting = "Hello, #{name}"
"Hello, John"
iex(3)> num = 15
15
iex(4)> results = "#{num} item(s) found."
"15 item(s) found."
```

Vérifiez si la chaîne contient la sous-chaîne

```
iex(1)> String.contains? "elixir of life", "of"
true
iex(2)> String.contains? "elixir of life", ["life", "death"]
true
```

```
iex(3)> String.contains? "elixir of life", ["venus", "mercury"]
false
```

Joindre des chaînes

Vous pouvez concaténer des chaînes dans Elixir en utilisant l'opérateur `<>` :

```
"Hello" <> "World"                          # => "HelloWorld"
```

Pour une `List` de chaînes, vous pouvez utiliser `Enum.join/2` :

```
Enum.join(["A", "few", "words"], " ")       # => "A few words"
```

Lire Cordes en ligne: https://riptutorial.com/fr/elixir/topic/2618/cordes

Chapitre 8: Correspondance de motif

Examples

Fonctions de correspondance de motif

```
#You can use pattern matching to run different
#functions based on which parameters you pass

#This example uses pattern matching to start,
#run, and end a recursive function

defmodule Counter do
    def count_to do
        count_to(100, 0) #No argument, init with 100
    end

    def count_to(counter) do
        count_to(counter, 0) #Initialize the recursive function
    end

    def count_to(counter, value) when value == counter do
        #This guard clause allows me to check my arguments against
        #expressions. This ends the recursion when the value matches
        #the number I am counting to.
        :ok
    end

    def count_to(counter, value) do
        #Actually do the counting
        IO.puts value
        count_to(counter, value + 1)
    end
end
```

Motif correspondant sur une carte

```
%{username: username} = %{username: "John Doe", id: 1}
# username == "John Doe"
```

```
%{username: username, id: 2} = %{username: "John Doe", id: 1}
** (MatchError) no match of right hand side value: %{id: 1, username: "John Doe"}
```

Correspondance de motif sur une liste

Vous pouvez également faire correspondre les structures de données Elixir telles que les listes.

Des listes

La correspondance sur une liste est assez simple.

```
[head | tail] = [1,2,3,4,5]
# head == 1
# tail == [2,3,4,5]
```

Cela fonctionne en faisant correspondre les premiers éléments (ou plus) de la liste à la gauche du `|` (pipe) et le reste de la liste à la variable de droite du `|` .

Nous pouvons également faire correspondre des valeurs spécifiques d'une liste:

```
[1,2 | tail] = [1,2,3,4,5]
# tail = [3,4,5]

[4 | tail] = [1,2,3,4,5]
** (MatchError) no match of right hand side value: [1, 2, 3, 4, 5]
```

Reliant plusieurs valeurs consécutives à gauche du `|` est également autorisé:

```
[a, b | tail] = [1,2,3,4,5]
# a == 1
# b == 2
# tail = [3,4,5]
```

Encore plus complexe - nous pouvons faire correspondre une valeur spécifique à une variable:

```
iex(11)> [a = 1 | tail] = [1,2,3,4,5]
# a == 1
```

Obtenir la somme d'une liste en utilisant la correspondance de modèle

```
defmodule Math do
  # We start of by passing the sum/1 function a list of numbers.
  def sum(numbers) do
    do_sum(numbers, 0)
  end

  # Recurse over the list when it contains at least one element.
  # We break the list up into two parts:
  #   head: the first element of the list
  #   tail: a list of all elements except the head
  # Every time this function is executed it makes the list of numbers
  # one element smaller until it is empty.
  defp do_sum([head|tail], acc) do
    do_sum(tail, head + acc)
  end

 # When we have reached the end of the list, return the accumulated sum
  defp do_sum([], acc), do: acc
end
```

Fonctions anonymes

```
f = fn
  {:a, :b} -> IO.puts "Tuple {:a, :b}"
```

```
  [] -> IO.puts "Empty list"
end

f.({:a, :b}) # Tuple {:a, :b}
f.([])       # Empty list
```

Tuples

```
{ a, b, c } = { "Hello", "World", "!" }

IO.puts a # Hello
IO.puts b # World
IO.puts c # !

# Tuples of different size won't match:

{ a, b, c } = { "Hello", "World" } # (MatchError) no match of right hand side value: {
"Hello", "World" }
```

Lecture d'un fichier

La correspondance de motif est utile pour une opération telle que la lecture de fichier qui renvoie un tuple.

Si le fichier `sample.txt` contient `This is a sample text` , alors:

```
{ :ok, file } = File.read("sample.txt")
# => {:ok, "This is a sample text"}

file
# => "This is a sample text"
```

Sinon, si le fichier n'existe pas:

```
{ :ok, file } = File.read("sample.txt")
# => ** (MatchError) no match of right hand side value: {:error, :enoent}

{ :error, msg } = File.read("sample.txt")
# => {:error, :enoent}
```

Fonctions anonymes d'appariement de motifs

```
fizzbuzz = fn
  (0, 0, _) -> "FizzBuzz"
  (0, _, _) -> "Fizz"
  (_, 0, _) -> "Buzz"
  (_, _, x) -> x
end

my_function = fn(n) ->
  fizzbuzz.(rem(n, 3), rem(n, 5), n)
end
```

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/elixir/topic/1602/correspondance-de-motif

Chapitre 9: Courant

Remarques

Les flux sont composables, énumérables paresseux.

En raison de leur paresse, les flux sont utiles lorsque vous travaillez avec des collections volumineuses (voire infinies). Lors du chaînage de nombreuses opérations avec `Enum` , des listes intermédiaires sont créées, tandis que `Stream` crée une recette de calculs exécutés ultérieurement.

Examples

Enchaînement de plusieurs opérations

`Stream` est particulièrement utile lorsque vous souhaitez exécuter plusieurs opérations sur une collection. C'est parce que `Stream` est paresseux et ne fait qu'une seule itération (alors `Enum` ferait plusieurs itérations, par exemple).

```
numbers = 1..100
|> Stream.map(fn(x) -> x * 2 end)
|> Stream.filter(fn(x) -> rem(x, 2) == 0 end)
|> Stream.take_every(3)
|> Enum.to_list

[2, 8, 14, 20, 26, 32, 38, 44, 50, 56, 62, 68, 74, 80, 86, 92, 98, 104, 110,
 116, 122, 128, 134, 140, 146, 152, 158, 164, 170, 176, 182, 188, 194, 200]
```

Ici, nous avons chaîné 3 opérations (`map` , `filter` et `take_every`), mais l'itération finale n'a été effectuée qu'après l' `Enum.to_list` de `Enum.to_list` .

Ce que fait `Stream` interne, c'est qu'il attend l'évaluation réelle. Avant cela, il crée une liste de toutes les fonctions, mais une fois que l'évaluation est nécessaire, elle parcourt la collection une fois, exécutant toutes les fonctions sur chaque élément. Cela le rend plus efficace que `Enum` , qui dans ce cas ferait 3 itérations, par exemple.

Lire Courant en ligne: https://riptutorial.com/fr/elixir/topic/2553/courant

Chapitre 10: Des listes

Syntaxe

- []
- [1, 2, 3, 4]
- [1, 2] ++ [3, 4] # -> [1,2,3,4]
- hd ([1, 2, 3, 4]) # -> 1
- tl ([1, 2, 3, 4]) # -> [2,3,4]
- [tête | queue]
- [1 | [2, 3, 4]] # -> [1,2,3,4]
- [1 | [2 | [3 | [4 | []]]]] -> [1,2,3,4]
- 'bonjour' = [? h,? e,? l,? l,? o]
- keyword_list = [a: 123, b: 456, c: 789]
- keyword_list [: a] # -> 123

Examples

Listes de mots clés

Les listes de mots-clés sont des listes où chaque élément de la liste est un tuple d'un atome suivi d'une valeur.

```
keyword_list = [{:a, 123}, {:b, 456}, {:c, 789}]
```

Une notation abrégée pour écrire des listes de mots clés est la suivante:

```
keyword_list = [a: 123, b: 456, c: 789]
```

Les listes de mots-clés sont utiles pour créer des structures de données de paires clé-valeur ordonnées, dans lesquelles plusieurs éléments peuvent exister pour une clé donnée.

Le premier élément d'une liste de mots clés pour une clé donnée peut être obtenu comme suit:

```
iex> keyword_list[:b]
456
```

Un cas d'utilisation des listes de mots clés pourrait être une séquence de tâches nommées à exécuter:

```
defmodule TaskRunner do
  def run_tasks(tasks) do
    # Call a function for each item in the keyword list.
    # Use pattern matching on each {:key, value} tuple in the keyword list
    Enum.each(tasks, fn
      {:delete, x} ->
```

```
        IO.puts("Deleting record " <> to_string(x) <> "...")
      {:add, value} ->
        IO.puts("Adding record \"" <> value <> "\"...")
      {:update, {x, value}} ->
        IO.puts("Setting record " <> to_string(x) <> " to \"" <> value <> "\"...")
    end)
  end
end
```

Ce code peut être appelé avec une liste de mots clés comme ceci:

```
iex> tasks = [
...>   add: "foo",
...>   add: "bar",
...>   add: "test",
...>   delete: 2,
...>   update: {1, "asdf"}
...> ]

iex> TaskRunner.run_tasks(tasks)
Adding record "foo"...
Adding record "bar"...
Adding record "test"...
Deleting record 2...
Setting record 1 to "asdf"...
```

Listes de char

Les chaînes dans Elixir sont des "binaires". Cependant, dans le code Erlang, les chaînes sont traditionnellement des "listes de caractères", donc lors de l'appel des fonctions Erlang, vous devrez peut-être utiliser des listes de caractères au lieu de chaînes Elixir classiques.

Alors que les chaînes régulières sont écrites à l'aide de guillemets `"` , les listes de caractères sont écrites à l'aide de guillemets simples `'` :

```
string = "Hello!"
char_list = 'Hello!'
```

Les listes de char sont simplement des listes d'entiers représentant les points de code de chaque caractère.

```
'hello' = [104, 101, 108, 108, 111]
```

Une chaîne peut être convertie en une liste de `to_charlist/1` avec `to_charlist/1` :

```
iex> to_charlist("hello")
'hello'
```

Et l'inverse peut être fait avec `to_string/1` :

```
iex> to_string('hello')
"hello"
```

Appeler une fonction Erlang et convertir la sortie en chaîne Elixir régulière:

```
iex> :os.getenv |> hd |> to_string
"PATH=/usr/local/bin:/usr/bin:/bin"
```

Cellules Contre

Les listes dans Elixir sont des listes liées. Cela signifie que chaque élément d'une liste consiste en une valeur, suivie d'un pointeur vers l'élément suivant de la liste. Ceci est implémenté dans Elixir en utilisant des cellules contre.

Les cellules cons sont des structures de données simples avec une valeur "gauche" et une valeur "droite", ou une "tête" et une "queue".

Un `|` Le symbole peut être ajouté avant le dernier élément d'une liste pour noter une liste (impropre) avec une tête et une queue données. Ce qui suit est une seule cellule avec `1` comme tête et `2` comme queue:

```
[1 | 2]
```

La syntaxe Elixir standard pour une liste est en réalité équivalente à l'écriture d'une chaîne de cellules imbriquées:

```
[1, 2, 3, 4] = [1 | [2 | [3 | [4 | []]]]]
```

La liste vide `[]` est utilisée comme queue d'une cellule contre pour représenter la fin d'une liste.

Toutes les listes dans Elixir sont équivalentes à la forme `[head | tail]` , où `head` est le premier élément de la liste et `tail` est le reste de la liste, moins la tête.

```
iex> [head | tail] = [1, 2, 3, 4]
[1, 2, 3, 4]
iex> head
1
iex> tail
[2, 3, 4]
```

Utiliser la `[head | tail]` notation est utile pour la correspondance de motif dans les fonctions récursives:

```
def sum([]), do: 0

def sum([head | tail]) do
  head + sum(tail)
end
```

Listes de cartographie

`map` est une fonction de programmation fonctionnelle qui, avec une liste et une fonction, renvoie

une nouvelle liste avec la fonction appliquée à chaque élément de cette liste. Dans Elixir, la fonction `map/2` trouve dans le module Enum .

```
iex> Enum.map([1, 2, 3, 4], fn(x) -> x + 1 end)
[2, 3, 4, 5]
```

Utiliser la syntaxe de capture alternative pour les fonctions anonymes:

```
iex> Enum.map([1, 2, 3, 4], &(&1 + 1))
[2, 3, 4, 5]
```

Se référant à une fonction avec une syntaxe de capture:

```
iex> Enum.map([1, 2, 3, 4], &to_string/1)
["1", "2", "3", "4"]
```

Chaînage des opérations de liste à l'aide de l'opérateur de tuyauterie:

```
iex> [1, 2, 3, 4]
...> |> Enum.map(&to_string/1)
...> |> Enum.map(&("Chapter " <> &1))
["Chapter 1", "Chapter 2", "Chapter 3", "Chapter 4"]
```

Liste des compréhensions

Elixir n'a pas de boucles. À la place des listes, il existe d'excellents modules `Enum` et `List` , mais il existe également des listes de compréhension.

Les compréhensions de liste peuvent être utiles pour:

- créer de nouvelles listes

```
iex(1)> for value <- [1, 2, 3], do: value + 1
[2, 3, 4]
```

- filtrage des listes, en utilisant `guard` expressions de `guard` , mais vous les utilisez sans mot `when` clé `when` .

```
iex(2)> odd? = fn x -> rem(x, 2) == 1 end
iex(3)> for value <- [1, 2, 3], odd?.(value), do: value
[1, 3]
```

- créer carte personnalisée, à l' aide `into` mot - clé:

```
iex(4)> for value <- [1, 2, 3], into: %{}, do: {value, value + 1}
%{1 => 2, 2=>3, 3 => 4}
```

Exemple combiné

```
iex(5)> for value <- [1, 2, 3], odd?.(value), into: %{}, do: {value, value * value}
%{1 => 1, 3 => 9}
```

Résumé

Liste de compréhension:

- utilise pour `for..do` syntaxe avec des gardes supplémentaires après les virgules et `into` mot-clé lors du retour d'une autre structure que les listes, c'est-à-dire. carte.
- dans d'autres cas, renvoyer de nouvelles listes
- ne supporte pas les accumulateurs
- ne peut pas arrêter le traitement lorsque certaines conditions sont remplies
- `guard` déclarations de `guard` doivent être en ordre après `for` et avant de `do` ou `into` symboles. L'ordre des symboles n'a pas d'importance

Selon ces contraintes, les compréhensions ne sont limitées que pour un usage simple. Dans les cas plus avancés, l'utilisation des fonctions des modules `Enum` et `List` serait la meilleure idée.

Différence de liste

```
iex> [1, 2, 3] -- [1, 3]
[2]
```

`--` supprime la première occurrence d'un élément de la liste de gauche pour chaque élément à droite.

Liste des membres

Utilisez `in` l' opérateur pour vérifier si un élément est un membre d'une liste.

```
iex> 2 in [1, 2, 3]
true
iex> "bob" in [1, 2, 3]
false
```

Conversion de listes en une carte

Utilisez `Enum.chunk/2` pour regrouper des éléments en sous-listes et `Map.new/2` pour le convertir en carte:

```
[1, 2, 3, 4, 5, 6]
|> Enum.chunk(2)
|> Map.new(fn [k, v] -> {k, v} end)
```

Donnerait:

```
%{1 => 2, 3 => 4, 5 => 6}
```

Lire Des listes en ligne: https://riptutorial.com/fr/elixir/topic/1279/des-listes

Chapitre 11: Doctests

Examples

introduction

Lorsque vous documentez votre code avec `@doc` , vous pouvez fournir des exemples de code comme ceci:

```
# myproject/lib/my_module.exs

defmodule MyModule do
  @doc """
  Given a number, returns `true` if the number is even, otherwise `false`.

  ## Example
    iex> MyModule.even?(2)
    true
    iex> MyModule.even?(3)
    false
  """
  def even?(number) do
    rem(number, 2) == 0
  end
end
```

Vous pouvez ajouter les exemples de code en tant que cas de test dans l'une de vos suites de tests:

```
# myproject/test/doc_test.exs

defmodule DocTest do
  use ExUnit.Case
  doctest MyModule
end
```

Ensuite, vous pouvez ensuite exécuter vos tests avec le `mix test` .

Générer de la documentation HTML basée sur doctest

La génération de la documentation étant basée sur le démarquage, vous devez faire 2 choses:

1 / Ecrivez votre doctest et donnez des exemples clairs de vos doctest pour améliorer la lisibilité (il est préférable de donner un titre, comme "des exemples" ou des "tests"). Lorsque vous écrivez vos tests, n'oubliez pas de donner 4 espaces à votre code de test pour qu'il soit formaté en tant que code dans la documentation HTML.

2 / Ensuite, entrez "mix docs" dans la console à la racine de votre projet elixir pour générer la documentation HTML dans le répertoire doc situé à la racine de votre projet elixir.

```
$> mix docs
```

Docteurs multilignes

Vous pouvez faire un doctest multiligne en utilisant "...>" pour les lignes qui suivent le premier

```
iex> Foo.Bar.somethingConditional("baz")
...>   |> case do
...>       {:ok, _} -> true
...>       {:error, _} -> false
...>      end
true
```

Lire Doctests en ligne: https://riptutorial.com/fr/elixir/topic/2708/doctests

Chapitre 12: Ecto

Examples

Ajouter un Ecto.Repo dans un programme d'élixir

Cela peut être fait en 3 étapes:

1. Vous devez définir un module elixir qui utilise Ecto.Repo et enregistrer votre application en tant que otp_app.

```
defmodule Repo do
  use Ecto.Repo, otp_app: :custom_app
end
```

2. Vous devez également définir une configuration pour le dépôt qui vous permettra de vous connecter à la base de données. Voici un exemple avec postgres.

```
config :custom_app, Repo,
   adapter: Ecto.Adapters.Postgres,
   database: "ecto_custom_dev",
   username: "postgres_dev",
   password: "postgres_dev",
   hostname: "localhost",
  # OR use a URL to connect instead
  url: "postgres://postgres_dev:postgres_dev@localhost/ecto_custom_dev"
```

3. Avant d'utiliser Ecto dans votre application, vous devez vous assurer que Ecto est démarré avant le démarrage de votre application. Cela peut être fait en enregistrant Ecto dans lib / custom_app.ex en tant que superviseur.

```
    def start(_type, _args) do
      import Supervisor.Spec

      children = [
       supervisor(Repo, [])
      ]

      opts = [strategy: :one_for_one, name: MyApp.Supervisor]
      Supervisor.start_link(children, opts)
    end
```

"et" clause dans un Repo.get_by / 3

Si vous avez un Ecto.Queryable, nommé Post, qui a un titre et une description.

Vous pouvez chercher le post avec le titre: "bonjour" et la description: "monde" en effectuant:

```
MyRepo.get_by(Post, [title: "hello", description: "world"])
```

Tout cela est possible car Repo.get_by attend en deuxième argument une liste de mots-clés.

Interroger avec des champs dynamiques

Pour interroger un champ dont le nom est contenu dans une variable, utilisez la fonction de champ .

```
some_field = :id
some_value = 10

from p in Post, where: field(p, ^some_field) == ^some_value
```

Ajouter des types de données personnalisés à la migration et au schéma

(De cette réponse)

L'exemple ci-dessous ajoute un type énuméré à une base de données postgres.

Tout d'abord, éditez le **fichier de migration** (créé avec `mix ecto.gen.migration`):

```
def up do
  # creating the enumerated type
  execute("CREATE TYPE post_status AS ENUM ('published', 'editing')")

  # creating a table with the column
  create table(:posts) do
    add :post_status, :post_status, null: false
  end
end

def down do
  drop table(:posts)
  execute("DROP TYPE post_status")
end
```

Deuxièmement, dans le **fichier modèle,** ajoutez un champ de type Elixir:

```
schema "posts" do
  field :post_status, :string
end
```

ou implémenter le comportement `Ecto.Type` .

Le paquet `ecto_enum` est un bon exemple et peut être utilisé comme modèle. Son utilisation est bien documentée sur sa **page github** .

Cette validation montre un exemple d'utilisation dans un projet Phoenix de l'ajout d'énum_ecto au projet et de l'utilisation du type énuméré dans les vues et les modèles.

Lire Ecto en ligne: https://riptutorial.com/fr/elixir/topic/6524/ecto

Chapitre 13: Erlang

Examples

Utiliser Erlang

Les modules Erlang sont disponibles sous forme d'atomes. Par exemple, le module de mathématiques Erlang est disponible sous forme de `:math` :

```
iex> :math.pi
3.141592653589793
```

Inspecter un module Erlang

Utilisez `module_info` sur les modules Erlang que vous souhaitez inspecter:

```
iex> :math.module_info
 [module: :math,
  exports: [pi: 0, module_info: 0, module_info: 1, pow: 2, atan2: 2, sqrt: 1,
   log10: 1, log2: 1, log: 1, exp: 1, erfc: 1, erf: 1, atanh: 1, atan: 1,
   asinh: 1, asin: 1, acosh: 1, acos: 1, tanh: 1, tan: 1, sinh: 1, sin: 1,
   cosh: 1, cos: 1],
  attributes: [vsn: [113168357788724588783826225069997113388]],
  compile: [options: [{:outdir,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/../ebin'},
    {:i,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/../include'},
    {:i,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-
18.3/lib/stdlib/src/../../kernel/include'},
    :warnings_as_errors, :debug_info], version: '6.0.2',
   time: {2016, 3, 16, 16, 40, 35},
   source: '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/math.erl'],
  native: false,
  md5: <<85, 35, 110, 210, 174, 113, 103, 228, 63, 252, 81, 27, 224, 15, 64,
    44>>]
```

Lire Erlang en ligne: https://riptutorial.com/fr/elixir/topic/2716/erlang

Chapitre 14: ExDoc

Examples

introduction

Pour générer de la documentation au format `HTML` partir des `@doc` et `@moduledoc` de votre code source, ajoutez `ex_doc` et un processeur de démarque, pour le moment ExDoc prend en charge Earmark , Pandoc , Hoedown et Cmark , en tant que dépendances dans votre fichier `mix.exs` :

```
# config/mix.exs

def deps do
  [{:ex_doc, "~> 0.11", only: :dev},
   {:earmark, "~> 0.1", only: :dev}]
end
```

Si vous souhaitez utiliser un autre processeur Markdown, vous pouvez trouver plus d'informations dans la section Outil Modification du Markdown .

Vous pouvez utiliser Markdown dans les `@doc` Elixir `@doc` et `@moduledoc` .

Ensuite, lancez `mix docs` .

Une chose à garder à l'esprit est que ExDoc autorise les paramètres de configuration, tels que:

```
  def project do
    [app: :my_app,
     version: "0.1.0-dev",
     name: "My App",
     source_url: "https://github.com/USER/APP",
     homepage_url: "http://YOUR_PROJECT_HOMEPAGE",
     deps: deps(),
     docs: [logo: "path/to/logo.png",
            output: "docs",
            main: "README",
            extra_section: "GUIDES",
            extras: ["README.md", "CONTRIBUTING.md"]]]
  end
```

Vous pouvez voir plus d'informations sur ces options de configuration avec l' `mix help docs`

Lire ExDoc en ligne: https://riptutorial.com/fr/elixir/topic/3582/exdoc

Chapitre 15: ExUnit

Examples

Affirmer des exceptions

Utilisez `assert_raise` pour tester si une exception a été `assert_raise` . `assert_raise` prend une exception et une fonction à exécuter.

```
test "invalid block size" do
  assert_raise(MerkleTree.ArgumentError, (fn() -> MerkleTree.new ["a", "b", "c"] end))
end
```

Enveloppez tout code que vous souhaitez tester dans une fonction anonyme et transmettez-le à `assert_raise` .

Lire ExUnit en ligne: https://riptutorial.com/fr/elixir/topic/3583/exunit

Chapitre 16: FAISCEAU

Examples

introduction

```
iex> :observer.start
:ok
```

`:observer.start` ouvre l'interface d'observateur de l'interface utilisateur graphique, en affichant la panne du processeur, l'utilisation de la mémoire et d'autres informations essentielles à la compréhension des modèles d'utilisation de vos applications.

Lire FAISCEAU en ligne: https://riptutorial.com/fr/elixir/topic/3587/faisceau

Chapitre 17: Installation

Examples

Installation de Fedora

```
dnf install erlang elixir
```

Installation OSX

Sous OS X et MacOS, Elixir peut être installé via les gestionnaires de paquets communs:

Homebrew

```
$ brew update
$ brew install elixir
```

Macports

```
 $ sudo port install elixir
```

Installation de Debian / Ubuntu

```
# Fetch and install package to setup access to the official APT repository
wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb
sudo dpkg -i erlang-solutions_1.0_all.deb

# Update package index
sudo apt-get update

# Install Erlang and Elixir
sudo apt-get install esl-erlang
sudo apt-get install elixir
```

Installation Gentoo / Funtoo

Elixir est disponible dans le dépôt principal des paquets.
Mettez à jour la liste des packages avant d'installer un package:

```
emerge --sync
```

Ceci est une installation en une étape:

```
emerge --ask dev-lang/elixir
```

Lire Installation en ligne: https://riptutorial.com/fr/elixir/topic/4208/installation

Chapitre 18: Joindre des chaînes

Examples

Utilisation de l'interpolation de chaîne

```
iex(1)> [x, y] = ["String1", "String2"]
iex(2)> "#{x} #{y}"
# "String1 String2"
```

Utiliser la liste IO

```
["String1", " ", "String2"] |> IO.iodata_to_binary
# "String1 String2"
```

Cette volonté donnera des performances boosts sous forme de chaînes non dupliquées en mémoire.
Méthode alternative:

```
iex(1)> IO.puts(["String1", " ", "String2"])
# String1 String2
```

Utiliser Enum.join

```
Enum.join(["String1", "String2"], " ")
# "String1 String2"
```

Lire Joindre des chaînes en ligne: https://riptutorial.com/fr/elixir/topic/9202/joindre-des-chaines

Chapitre 19: Les constantes

Remarques

Voici donc une analyse récapitulative que j'ai effectuée sur la base des méthodes répertoriées dans Comment définir des constantes dans les modules Elixir? . Je le poste pour deux raisons:

- La plupart des documents d'Elixir sont assez complets, mais j'ai trouvé cette décision architecturale clé manquant de conseils - alors je l'aurais demandé comme sujet.
- Je voulais avoir un peu de visibilité et des commentaires des autres sur le sujet.
- Je voulais également tester le nouveau flux de production de la documentation SO. ;)

J'ai également téléchargé l'intégralité du code sur le concept GitHub repo elixir-constants .

Examples

Constantes de portée de module

```
defmodule MyModule do
  @my_favorite_number 13
  @use_snake_case "This is a string (use double-quotes)"
end
```

Celles-ci ne sont accessibles que depuis ce module.

Constantes comme fonctions

Déclarer:

```
defmodule MyApp.ViaFunctions.Constants do
  def app_version, do: "0.0.1"
  def app_author, do: "Felix Orr"
  def app_info, do: [app_version, app_author]
  def bar, do: "barrific constant in function"
end
```

Consommer avec exigent:

```
defmodule MyApp.ViaFunctions.ConsumeWithRequire do
  require MyApp.ViaFunctions.Constants

  def foo() do
    IO.puts MyApp.ViaFunctions.Constants.app_version
    IO.puts MyApp.ViaFunctions.Constants.app_author
    IO.puts inspect MyApp.ViaFunctions.Constants.app_info
  end

  # This generates a compiler error, cannot invoke `bar/0` inside a guard.
  # def foo(_bar) when is_bitstring(bar) do
```

```
  #   IO.puts "We just used bar in a guard: #{bar}"
  # end
end
```

À consommer avec importation:

```
defmodule MyApp.ViaFunctions.ConsumeWithImport do
  import MyApp.ViaFunctions.Constants

  def foo() do
    IO.puts app_version
    IO.puts app_author
    IO.puts inspect app_info
  end
end
```

Cette méthode permet de réutiliser les constantes sur les projets, mais elles ne seront pas utilisables dans les fonctions de garde qui nécessitent des constantes de compilation.

Constantes via des macros

Déclarer:

```
defmodule MyApp.ViaMacros.Constants do
  @moduledoc """
  Apply with `use MyApp.ViaMacros.Constants, :app` or `import MyApp.ViaMacros.Constants,
:app`.

  Each constant is private to avoid ambiguity when importing multiple modules
  that each have their own copies of these constants.
  """

  def app do
    quote do
      # This method allows sharing module constants which can be used in guards.
      @bar "barrific module constant"
      defp app_version, do: "0.0.1"
      defp app_author, do: "Felix Orr"
      defp app_info, do: [app_version, app_author]
    end
  end

  defmacro __using__(which) when is_atom(which) do
    apply(__MODULE__, which, [])
  end
end
```

À consommer avec `use` :

```
defmodule MyApp.ViaMacros.ConsumeWithUse do
  use MyApp.ViaMacros.Constants, :app

  def foo() do
    IO.puts app_version
    IO.puts app_author
    IO.puts inspect app_info
```

```
  end

  def foo(_bar) when is_bitstring(@bar) do
    IO.puts "We just used bar in a guard: #{@bar}"
  end
end
```

Cette méthode vous permet d'utiliser `@some_constant` intérieur des gardes. Je ne suis même pas sûr que les fonctions seraient strictement nécessaires.

Lire Les constantes en ligne: https://riptutorial.com/fr/elixir/topic/6614/les-constantes

Chapitre 20: Les fonctions

Examples

Fonctions anonymes

Dans Elixir, une pratique courante consiste à utiliser des fonctions anonymes. Créer une fonction anonyme est simple:

```
iex(1)> my_func = fn x -> x * 2 end
#Function<6.52032458/1 in :erl_eval.expr/5>
```

La syntaxe générale est la suivante:

```
fn args -> output end
```

Pour plus de lisibilité, vous pouvez mettre des parenthèses autour des arguments:

```
iex(2)> my_func = fn (x, y) -> x*y end
#Function<12.52032458/2 in :erl_eval.expr/5>
```

Pour appeler une fonction anonyme, appelez-la par le nom attribué et ajoutez `.` entre le nom et les arguments.

```
iex(3)>my_func.(7, 5)
35
```

Il est possible de déclarer des fonctions anonymes sans arguments:

```
iex(4)> my_func2 = fn -> IO.puts "hello there" end
iex(5)> my_func2.()
hello there
:ok
```

Utilisation de l'opérateur de capture

Pour rendre les fonctions anonymes plus concises, vous pouvez utiliser l' **opérateur de capture** `&` . Par exemple, au lieu de:

```
iex(5)> my_func = fn (x) -> x*x*x end
```

Tu peux écrire:

```
iex(6)> my_func = &(&1*&1*&1)
```

Avec plusieurs paramètres, utilisez le nombre correspondant à chaque argument, en partant de `1` :

```
iex(7)> my_func = fn (x, y) -> x + y end

iex(8)> my_func = &(&1 + &2)    # &1 stands for x and &2 stands for y

iex(9)> my_func.(4, 5)
9
```

Plusieurs corps

Une fonction anonyme peut également avoir plusieurs corps (à la suite d' une correspondance de modèle):

```
my_func = fn
  param1 -> do_this
  param2 -> do_that
end
```

Lorsque vous appelez une fonction avec plusieurs corps, Elixir tente de faire correspondre les paramètres fournis avec le corps de la fonction approprié.

Listes de mots clés en tant que paramètres de fonction

Utilisez des listes de mots-clés pour les paramètres de style "options" contenant plusieurs paires clé-valeur:

```
def myfunc(arg1, opts \\ []) do
  # Function body
end
```

Nous pouvons appeler la fonction ci-dessus ainsi:

```
iex> myfunc "hello", pizza: true, soda: false
```

ce qui équivaut à:

```
iex> myfunc("hello", [pizza: true, soda: false])
```

Les valeurs d'argument sont disponibles respectivement sous la forme `opts.pizza` et `opts.soda` . Vous pouvez également utiliser des atomes: `opts[:pizza]` et `opts[:soda]` .

Fonctions nommées et fonctions privées

Fonctions nommées

```
defmodule Math do
    # one way
```

```
    def add(a, b) do
        a + b
    end

    # another way
    def subtract(a, b), do: a - b
end

iex> Math.add(2, 3)
5
:ok
iex> Math.subtract(5, 2)
3
:ok
```

Fonctions Privées

```
defmodule Math do
    def sum(a, b) do
        add(a, b)
    end

    # Private Function
    defp add(a, b) do
        a + b
    end
end

iex> Math.add(2, 3)
** (UndefinedFunctionError) undefined function Math.add/2
Math.add(3, 4)
iex> Math.sum(2, 3)
5
```

Correspondance de motif

Elixir correspond à un appel de fonction à son corps en fonction de la valeur de ses arguments.

```
defmodule Math do
    def factorial(0): do: 1
    def factorial(n): do: n * factorial(n - 1)
end
```

Ici, la factorielle des nombres positifs correspond à la deuxième clause, tandis que la `factorial(0)` correspond à la première. (en ignorant les nombres négatifs pour des raisons de simplicité). Elixir essaie de faire correspondre les fonctions de haut en bas. Si la deuxième fonction est écrite au-dessus du premier, nous obtiendrons un résultat inattendu à mesure qu'il se dirigera vers une récursion sans fin. Parce que `factorial(0)` correspond à `factorial(n)`

Clauses de garde

Les clauses de garde nous permettent de vérifier les arguments avant d'exécuter la fonction. Les clauses de garde sont généralement préférées à `if` et `cond` raison de leur lisibilité, et pour faciliter une certaine technique d'optimisation pour le compilateur. La première définition de fonction où

tous les gardes correspondent est exécutée.

Voici un exemple d'implémentation de la fonction factorielle à l'aide de gardes et de correspondance de modèle.

```
defmodule Math do
    def factorial(0), do: 1
    def factorial(n) when n > 0: do: n * factorial(n - 1)
end
```

Le premier motif correspond si (et seulement si) l'argument est `0` . Si l'argument n'est pas `0` , la correspondance du modèle échoue et la fonction suivante ci-dessous est cochée.

Cette deuxième définition de fonction a une clause de garde: `when n > 0` . Cela signifie que cette fonction ne correspond que si l'argument `n` est supérieur à `0` . Après tout, la fonction factorielle mathématique n'est pas définie pour les entiers négatifs.

Si aucune définition de fonction (y compris leur correspondance de modèle et les clauses de garde) ne correspond, une `FunctionClauseError` sera déclenchée. Cela se produit pour cette fonction lorsque nous passons un nombre négatif comme argument, car il n'est pas défini pour les nombres négatifs.

Notez que cette `FunctionClauseError` n'est pas une erreur. Renvoyer `-1` ou `0` ou une autre "valeur d'erreur" comme cela est courant dans certaines autres langues cacherait le fait que vous avez appelé une fonction indéfinie, en masquant la source de l'erreur, créant éventuellement un énorme bogue douloureux pour un futur développeur.

Paramètres par défaut

Vous pouvez transmettre les paramètres par défaut à toute fonction nommée en utilisant la syntaxe suivante: `param \\ value` :

```
defmodule Example do
    def func(p1, p2 \\ 2) do
        IO.inspect [p1, p2]
    end
end

Example.func("a")     # => ["a", 2]
Example.func("b", 4) # => ["b", 4]
```

Fonctions de capture

Utilisez `&` pour capturer les fonctions des autres modules. Vous pouvez utiliser les fonctions capturées directement en tant que paramètres de fonction ou dans des fonctions anonymes.

```
Enum.map(list, fn(x) -> String.capitalize(x) end)
```

Peut être rendu plus concis en utilisant `&` :

```
Enum.map(list, &String.capitalize(&1))
```

La capture de fonctions sans passer d'argument nécessite de spécifier explicitement son arité, par exemple `&String.capitalize/1` :

```
defmodule Bob do
  def say(message, f \\ &String.capitalize/1) do
    f.(message)
  end
end
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/elixir/topic/2442/les-fonctions

Chapitre 21: Les nœuds

Examples

Liste tous les nœuds visibles dans le système

```
iex(bob@127.0.0.1)> Node.list
[:"frank@127.0.0.1"]
```

Connexion des nœuds sur la même machine

Démarrez deux nœuds nommés dans deux fenêtres de terminal:

```
>iex --name bob@127.0.0.1
iex(bob@127.0.0.1)>
>iex --name frank@127.0.0.1
iex(frank@127.0.0.1)>
```

Connectez deux nœuds en demandant à un nœud de se connecter:

```
iex(bob@127.0.0.1)> Node.connect :"frank@127.0.0.1"
true
```

Les deux nœuds sont maintenant connectés et conscients l'un de l'autre:

```
iex(bob@127.0.0.1)> Node.list
[:"frank@127.0.0.1"]
iex(frank@127.0.0.1)> Node.list
[:"bob@127.0.0.1"]
```

Vous pouvez exécuter du code sur d'autres nœuds:

```
iex(bob@127.0.0.1)> greet = fn() -> IO.puts("Hello from #{inspect(Node.self)}") end
iex(bob@127.0.0.1)> Node.spawn(:"frank@127.0.0.1", greet)
#PID<9007.74.0>
Hello from :"frank@127.0.0.1"
:ok
```

Connexion des nœuds sur des machines différentes

Démarrer un processus nommé sur une adresse IP:

```
$ iex --name foo@10.238.82.82 --cookie chocolate
iex(foo@10.238.82.82)> Node.ping :"bar@10.238.82.85"
:pong
iex(foo@10.238.82.82)> Node.list
[:"bar@10.238.82.85"]
```

Démarrez un autre processus nommé sur une adresse IP différente:

```
$ iex --name bar@10.238.82.85 --cookie chocolate
iex(bar@10.238.82.85)> Node.list
[:"foo@10.238.82.82"]
```

Lire Les nœuds en ligne: https://riptutorial.com/fr/elixir/topic/2065/les-nouds

Chapitre 22: Les opérateurs

Examples

L'opérateur de tuyaux

Le Pipe Operator `|>` prend le résultat d'une expression à gauche et l'utilise comme premier paramètre d'une fonction à droite.

```
expression |> function
```

Utilisez l'opérateur Pipe pour enchaîner les expressions et documenter visuellement le déroulement d'une série de fonctions.

Considérer ce qui suit:

```
Oven.bake(Ingredients.Mix([:flour, :cocoa, :sugar, :milk, :eggs, :butter]), :temperature)
```

Dans l'exemple, `Oven.bake` arrive avant `Ingredients.mix` , mais il est exécuté en dernier. En outre, il peut ne pas être évident que `:temperature` est un paramètre de `Oven.bake`

Réécriture de cet exemple à l'aide de l'opérateur Pipe:

```
[:flour, :cocoa, :sugar, :milk, :eggs, :butter]
|> Ingredients.mix
|> Oven.bake(:temperature)
```

donne le même résultat, mais l'ordre d'exécution est plus clair. De plus, il est clair que `:temperature` est un paramètre de l'appel `Oven.bake` .

Notez que lors de l'utilisation de l'opérateur Pipe, le premier paramètre de chaque fonction est déplacé avant l'opérateur Pipe, de sorte que la fonction appelée semble avoir un paramètre de moins. Par exemple:

```
Enum.each([1, 2, 3], &(&1+1)) # produces [2, 3, 4]
```

est le même que:

```
[1, 2, 3]
|> Enum.each(&(&1+1))
```

Conducteur et parenthèses

Les parenthèses sont nécessaires pour éviter toute ambiguïté:

```
foo 1 |> bar 2 |> baz 3
```

Devrait être écrit comme:

```
foo(1) |> bar(2) |> baz(3)
```

opérateurs booléens

Il existe deux types d'opérateurs booléens dans Elixir:

- les opérateurs booléens (ils s'attendent à `true` ou `false` comme premier argument)

```
x or y      # true if x is true, otherwise y

x and y     # false if x is false, otherwise y

not x       # false if x is true, otherwise true
```

Tous les opérateurs booléens déclencheront `ArgumentError` si le premier argument ne sera pas strictement une valeur booléenne, ce qui signifie seulement `true` ou `false` (`nil` n'est pas booléen).

```
iex(1)> false and 1 # return false
iex(2)> false or 1 # return 1
iex(3)> nil and 1 # raise (ArgumentError) argument error: nil
```

- opérateurs booléens décontractés (travail avec n'importe quel type, tout ce que ni `false` ni `nil` n'est considéré comme `true`)

```
x || y     # x if x is true, otherwise y

x && y     # y if x is true, otherwise false

!x         # false if x is true, otherwise true
```

Opérateur `||` retournera toujours le premier argument si c'est vrai (Elixir traite tout sauf `nil` et `false` pour être vrai dans les comparaisons), sinon retournera le second.

```
iex(1)> 1 || 3 # return 1, because 1 is truthy
iex(2)> false || 3 # return 3
iex(3)> 3 || false # return 3
iex(4)> false || nil # return nil
iex(5)> nil || false # return false
```

L'opérateur `&&` renverra toujours le deuxième argument si c'est vrai. Sinon, reviendra respectivement aux arguments, `false` ou `nil` .

```
iex(1)> 1 && 3 # return 3, first argument is truthy
iex(2)> false && 3 # return false
iex(3)> 3 && false # return false
iex(4)> 3 && nil # return nil
iex(5)> false && nil # return false
iex(6)> nil && false # return nil
```

Les deux `&&` et `||` sont des opérateurs de court-circuit. Ils n'exécutent que le côté droit si le côté gauche n'est pas suffisant pour déterminer le résultat.

Opérateur `!` renverra une valeur booléenne de négation du terme courant:

```
iex(1)> !2 # return false
iex(2)> !false # return true
iex(3)> !"Test" # return false
iex(4)> !nil # return true
```

Un moyen simple d'obtenir une valeur booléenne du terme sélectionné est de simplement doubler cet opérateur:

```
iex(1)> !!true # return true
iex(2)> !!"Test" # return true
iex(3)> !!nil # return false
iex(4)> !!false # return false
```

Opérateurs de comparaison

Égalité:

- valeur égalité `x == y` (`1 == 1.0 # true`)
- valeur inégalité `x == y` (`1 != 1.0 # false`)
- égalité stricte `x === y` (`1 === 1.0 # false`)
- Inégalité stricte `x === y` (`1 !== 1.0 # true`)

Comparaison:

- `x > y`
- `x >= y`
- `x < y`
- `x <= y`

Si les types sont compatibles, la comparaison utilise l'ordre naturel. Sinon, il existe une règle de comparaison des types généraux:

```
number < atom < reference < function < port < pid < tuple < map < list < binary
```

Rejoindre les opérateurs

Vous pouvez joindre (concaténer) des binaires (y compris des chaînes) et des listes:

```
iex(1)> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]

iex(2)> [1, 2, 3, 4, 5] -- [1, 3]
[2, 4, 5]

iex(3)> "qwe" <> "rty"
"qwerty"
```

Opérateur 'In'

`in` operator vous permet de vérifier si une liste ou une plage contient un élément:

```
iex(4)> 1 in [1, 2, 3, 4]
true

iex(5)> 0 in (1..5)
false
```

Lire Les opérateurs en ligne: https://riptutorial.com/fr/elixir/topic/1161/les-operateurs

Chapitre 23: Manipulation d'Etat à Elixir

Examples

Gérer un morceau d'état avec un agent

Le moyen le plus simple d'emballer et d'accéder à un état est l' `Agent` . Le module permet de générer un processus qui conserve une structure de données arbitraire et permet d'envoyer des messages pour lire et mettre à jour cette structure. Grâce à cela, l'accès à la structure est automatiquement sérialisé, car le processus ne gère qu'un seul message à la fois.

```
iex(1)> {:ok, pid} = Agent.start_link(fn -> :initial_value end)
{:ok, #PID<0.62.0>}
iex(2)> Agent.get(pid, &(&1))
:initial_value
iex(3)> Agent.update(pid, fn(value) -> {value, :more_data} end)
:ok
iex(4)> Agent.get(pid, &(&1))
{:initial_value, :more_data}
```

Lire Manipulation d'Etat à Elixir en ligne: https://riptutorial.com/fr/elixir/topic/6596/manipulation-d-etat-a-elixir

Chapitre 24: Meilleur débogage avec IO.inspect et les étiquettes

Introduction

`IO.inspect` est très utile lorsque vous essayez de déboguer vos chaînes d'appels de méthode. Cela peut être compliqué si vous l'utilisez trop souvent.

Depuis Elixir 1.4.0, l'option d' `label` d' `IO.inspect` peut aider

Remarques

Ne fonctionne qu'avec Elixir 1.4+, mais je ne peux pas encore le marquer.

Examples

Sans étiquettes

```
url
  |> IO.inspect
  |> HTTPoison.get!
  |> IO.inspect
  |> Map.get(:body)
  |> IO.inspect
  |> Poison.decode!
  |> IO.inspect
```

Cela se traduira par beaucoup de sortie sans contexte:

```
"https://jsonplaceholder.typicode.com/posts/1"
%HTTPoison.Response{body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere
repellat provident occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et
suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut
quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}",
 headers: [{"Date", "Thu, 05 Jan 2017 14:29:59 GMT"},
  {"Content-Type", "application/json; charset=utf-8"},
  {"Content-Length", "292"}, {"Connection", "keep-alive"},
  {"Set-Cookie",
   "__cfduid=d56d1be0a544fcbdbb262fee9477600c51483626599; expires=Fri, 05-Jan-18 14:29:59 GMT;
path=/; domain=.typicode.com; HttpOnly"},
  {"X-Powered-By", "Express"}, {"Vary", "Origin, Accept-Encoding"},
  {"Access-Control-Allow-Credentials", "true"},
  {"Cache-Control", "public, max-age=14400"}, {"Pragma", "no-cache"},
  {"Expires", "Thu, 05 Jan 2017 18:29:59 GMT"},
  {"X-Content-Type-Options", "nosniff"},
  {"Etag", "W/\"124-yv65LoT2uMHrpn06wNpAcQ\""}, {"Via", "1.1 vegur"},
  {"CF-Cache-Status", "HIT"}, {"Server", "cloudflare-nginx"},
  {"CF-RAY", "31c7a025e94e2d41-TXL"}], status_code: 200}
"{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere repellat provident
```

```
occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et suscipit\\nsuscipit
recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum
rerum est autem sunt rem eveniet architecto\"\n}"
%{"body" => "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit
molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto",
  "id" => 1,
  "title" => "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "userId" => 1}
```

Avec des étiquettes

L'utilisation de l'option `label` pour ajouter du contexte peut aider beaucoup:

```
url
  |> IO.inspect(label: "url")
  |> HTTPoison.get!
  |> IO.inspect(label: "raw http resonse")
  |> Map.get(:body)
  |> IO.inspect(label: "raw body")
  |> Poison.decode!
  |> IO.inspect(label: "parsed body")

url: "https://jsonplaceholder.typicode.com/posts/1"
raw http resonse: %HTTPoison.Response{body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\":
\"sunt aut facere repellat provident occaecati excepturi optio reprehenderit\",\n  \"body\":
\"quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit
molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}",
 headers: [{"Date", "Thu, 05 Jan 2017 14:33:06 GMT"},
  {"Content-Type", "application/json; charset=utf-8"},
  {"Content-Length", "292"}, {"Connection", "keep-alive"},
  {"Set-Cookie",
   "__cfduid=d22d817e48828169296605d27270af7e81483626786; expires=Fri, 05-Jan-18 14:33:06 GMT;
path=/; domain=.typicode.com; HttpOnly"},
  {"X-Powered-By", "Express"}, {"Vary", "Origin, Accept-Encoding"},
  {"Access-Control-Allow-Credentials", "true"},
  {"Cache-Control", "public, max-age=14400"}, {"Pragma", "no-cache"},
  {"Expires", "Thu, 05 Jan 2017 18:33:06 GMT"},
  {"X-Content-Type-Options", "nosniff"},
  {"Etag", "W/\"124-yv65LoT2uMHrpn06wNpAcQ\""}, {"Via", "1.1 vegur"},
  {"CF-Cache-Status", "HIT"}, {"Server", "cloudflare-nginx"},
  {"CF-RAY", "31c7a4b8ae042d77-TXL"}], status_code: 200}
raw body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere repellat
provident occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et
suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut
quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}"
parsed body: %{"body" => "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet
architecto",
  "id" => 1,
  "title" => "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "userId" => 1}
```

Lire Meilleur débogage avec IO.inspect et les étiquettes en ligne:
https://riptutorial.com/fr/elixir/topic/8725/meilleur-debogage-avec-io-inspect-et-les-etiquettes

Chapitre 25: Mélanger

Examples

Créer une tâche de mixage personnalisée

```
# lib/mix/tasks/mytask.ex
defmodule Mix.Tasks.MyTask do
  use Mix.Task

  @shortdoc "A simple mix task"
  def run(_) do
    IO.puts "YO!"
  end
end
```

Compiler et exécuter:

```
$ mix compile
$ mix my_task
"YO!"
```

Tâche de mixage personnalisée avec des arguments de ligne de commande

Dans une implémentation de base, le module de tâche doit définir une `run/1` qui prend une liste d'arguments. Par exemple, `def run(args) do ... end`

```
defmodule Mix.Tasks.Example_Task do
  use Mix.Task

  @shortdoc "Example_Task prints hello + its arguments"
  def run(args) do
    IO.puts "Hello #{args}"
  end
end
```

Compiler et exécuter:

```
$ mix example_task world
"hello world"
```

Alias

Elixir vous permet d'ajouter des alias pour vos commandes de mixage. Chose cool si vous voulez vous épargner de la frappe.

Ouvrez `mix.exs` dans votre projet Elixir.

Tout d'abord, ajoutez la fonction `aliases/0` à la liste de mots clés renvoyée par la fonction `project` .

Ajouter `()` à la fin de la fonction alias empêchera le compilateur de lancer un avertissement.

```
def project do
  [app: :my_app,
   ...
   aliases: aliases()]
end
```

Ensuite, définissez votre fonction `aliases/0` (par exemple en bas de votre fichier `mix.exs`).

```
...

defp aliases do
  [go: "phoenix.server",
   trident: "do deps.get, compile, go"]
end
```

Vous pouvez maintenant utiliser `$ mix go` pour exécuter votre serveur Phoenix (si vous utilisez une application Phoenix). Et utilisez `$ mix trident` pour dire à mix de récupérer toutes les dépendances, compiler et exécuter le serveur.

Obtenir de l'aide sur les tâches de mixage disponibles

Pour répertorier les tâches de mixage disponibles, utilisez:

```
mix help
```

Pour obtenir de l'aide sur une tâche spécifique, utilisez la tâche d'aide de `mix help task` par exemple:

```
mix help cmd
```

Lire Mélanger en ligne: https://riptutorial.com/fr/elixir/topic/3585/melanger

Chapitre 26: Métaprogrammation

Examples

Générer des tests au moment de la compilation

```
defmodule ATest do
  use ExUnit.Case

  [{1, 2, 3}, {10, 20, 40}, {100, 200, 300}]
  |> Enum.each(fn {a, b, c} ->
    test "#{a} + #{b} = #{c}" do
      assert unquote(a) + unquote(b) = unquote(c)
    end
  end)
end
```

Sortie:

```
.

  1) test 10 + 20 = 40 (Test.Test)
     test.exs:6
     match (=) failed
     code: 10 + 20 = 40
     rhs:  40
     stacktrace:
       test.exs:7

.

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
3 tests, 1 failure
```

Lire Métaprogrammation en ligne: https://riptutorial.com/fr/elixir/topic/4069/metaprogrammation

Chapitre 27: Modules

Remarques

Noms de module

Dans Elixir, les noms de module tels que `IO` ou `String` sont que des atomes sous le capot et sont convertis au `:"Elixir.ModuleName"` au moment de la compilation.

```
iex(1)> is_atom(IO)
true
iex(2)> IO == :"Elixir.IO"
true
```

Examples

Liste des fonctions ou des macros d'un module

La fonction `__info__/1` prend l'un des atomes suivants:

- `:functions` - Retourne une liste de mots clés des fonctions publiques avec leurs arités
- `:macros` - Retourne une liste de mots clés de macros publiques avec leurs arités

Pour lister les fonctions du module `Kernel` :

```
iex> Kernel.__info__ :functions
[!=: 2, !==: 2, *: 2, +: 1, +: 2, ++: 2, -: 1, -: 2, --: 2, /: 2, <: 2, <=: 2,
 ==: 2, ===: 2, =~: 2, >: 2, >=: 2, abs: 1, apply: 2, apply: 3, binary_part: 3,
 bit_size: 1, byte_size: 1, div: 2, elem: 2, exit: 1, function_exported?: 3,
 get_and_update_in: 3, get_in: 2, hd: 1, inspect: 1, inspect: 2, is_atom: 1,
 is_binary: 1, is_bitstring: 1, is_boolean: 1, is_float: 1, is_function: 1,
 is_function: 2, is_integer: 1, is_list: 1, is_map: 1, is_number: 1, is_pid: 1,
 is_port: 1, is_reference: 1, is_tuple: 1, length: 1, macro_exported?: 3,
 make_ref: 0, ...]
```

Remplacez le `Kernel` par n'importe quel module de votre choix.

Utiliser des modules

Les modules ont quatre mots-clés associés pour les utiliser dans d'autres modules: `alias` , `import` , `use` et `require` .

`alias` enregistrera un module sous un nom différent (généralement plus court):

```
defmodule MyModule do
  # Will make this module available as `CoolFunctions`
  alias MyOtherModule.CoolFunctions
  # Or you can specify the name to use
```

```
  alias MyOtherModule.CoolFunctions, as: CoolFuncs
end
```

`import` rend toutes les fonctions du module disponibles sans nom devant elles:

```
defmodule MyModule do
  import Enum
  def do_things(some_list) do
    # No need for the `Enum.` prefix
    join(some_list, " ")
  end
end
```

`use` permet à un module d'injecter du code dans le module actuel - cela se fait généralement dans le cadre d'une structure qui crée ses propres fonctions pour que votre module confirme son comportement.

`require` des macros de chargement du module pour pouvoir les utiliser.

Déléguer des fonctions à un autre module

Utilisez `defdelegate` pour définir des fonctions qui délèguent aux fonctions du même nom définies dans un autre module:

```
defmodule Math do
  defdelegate pi, to: :math
end


iex> Math.pi
3.141592653589793
```

Lire Modules en ligne: https://riptutorial.com/fr/elixir/topic/2721/modules

Chapitre 28: Obtenir de l'aide dans la console IEx

Introduction

IEx donne accès à la documentation d'Elixir. Lorsque Elixir est installé sur votre système, vous pouvez lancer IEx, par exemple avec la commande `iex` dans un terminal. Ensuite, tapez la commande `h` sur la ligne de commande IEx suivie du nom de la fonction précédé du nom du module, par exemple `h List.foldr`

Examples

Liste des modules et fonctions d'Elixir

Pour obtenir la liste des modules Elixir, tapez simplement

```
h Elixir.[TAB]
```

Appuyer sur [TAB] pour compléter automatiquement les noms des modules et des fonctions. Dans ce cas, il répertorie tous les modules. Pour trouver toutes les fonctions d'un module, par exemple, utiliser la `List`

```
h List.[TAB]
```

Lire Obtenir de l'aide dans la console IEx en ligne:
https://riptutorial.com/fr/elixir/topic/10780/obtenir-de-l-aide-dans-la-console-iex

Chapitre 29: Optimisation

Examples

Toujours mesurer en premier!

Ce sont des conseils généraux qui améliorent en général les performances. Si votre code est lent, il est toujours important de le profiler pour déterminer quelles pièces sont lentes. Deviner n'est **jamais** suffisant. Améliorer la vitesse d'exécution de quelque chose qui ne prend que 1% du temps d'exécution ne vaut probablement pas la peine. Cherchez les puits de temps.

Pour obtenir des chiffres assez précis, assurez-vous que le code que vous optimisez est exécuté pendant au moins une seconde lors du profilage. Si vous consacrez 10% du temps d'exécution à cette fonction, assurez-vous que l'exécution complète du programme dure au moins 10 secondes et assurez-vous que vous pouvez exécuter les mêmes données exactes via le code plusieurs fois pour obtenir des nombres reproductibles.

ExProf est simple pour commencer.

Lire Optimisation en ligne: https://riptutorial.com/fr/elixir/topic/6062/optimisation

Chapitre 30: Polymorphisme chez Elixir

Introduction

Le polymorphisme consiste à fournir une interface unique à des entités de types différents. Fondamentalement, il permet à différents types de données de répondre à la même fonction. Donc, les mêmes fonctions de forme pour différents types de données pour accomplir le même comportement. Le langage Elixir a des `protocols` pour implémenter le polymorphisme de manière propre.

Remarques

Si vous souhaitez couvrir tous les types de données, vous pouvez définir une implémentation pour `Any` type de données. Enfin, si vous avez le temps, vérifiez le code source de Enum et String.Char , qui sont de bons exemples de polymorphisme dans le noyau Elixir.

Examples

Polymorphisme Avec Protocoles

Implémentons un protocole de base qui convertit les températures Kelvin et Fahrenheit en Celsius.

```
defmodule Kelvin do
  defstruct name: "Kelvin", symbol: "K", degree: 0
end

defmodule Fahrenheit do
  defstruct name: "Fahrenheit", symbol: "°F", degree: 0
end

defmodule Celsius do
  defstruct name: "Celsius", symbol: "°C", degree: 0
end

defprotocol Temperature do
  @doc """
  Convert Kelvin and Fahrenheit to Celsius degree
  """
  def to_celsius(degree)
end

defimpl Temperature, for: Kelvin do
  @doc """
  Deduct 273.15
  """
  def to_celsius(kelvin) do
    celsius_degree = kelvin.degree - 273.15
    %Celsius{degree: celsius_degree}
  end
```

```
end

defimpl Temperature, for: Fahrenheit do
  @doc """
  Deduct 32, then multiply by 5, then divide by 9
  """
  def to_celsius(fahrenheit) do
    celsius_degree = (fahrenheit.degree - 32) * 5 / 9
    %Celsius{degree: celsius_degree}
  end
end
```

Maintenant, nous avons implémenté nos convertisseurs pour les types Kelvin et Fahrenheit. Faisons quelques conversions:

```
iex> fahrenheit = %Fahrenheit{degree: 45}
%Fahrenheit{degree: 45, name: "Fahrenheit", symbol: "°F"}
iex> celsius = Temperature.to_celsius(fahrenheit)
%Celsius{degree: 7.22, name: "Celsius", symbol: "°C"}
iex> kelvin = %Kelvin{degree: 300}
%Kelvin{degree: 300, name: "Kelvin", symbol: "K"}
iex> celsius = Temperature.to_celsius(kelvin)
%Celsius{degree: 26.85, name: "Celsius", symbol: "°C"}
```

Essayons de convertir tout autre type de données sans implémentation pour la fonction `to_celsius` :

```
iex> Temperature.to_celsius(%{degree: 12})
** (Protocol.UndefinedError) protocol Temperature not implemented for %{degree: 12}
    iex:11: Temperature.impl_for!/1
    iex:15: Temperature.to_celsius/1
```

Lire Polymorphisme chez Elixir en ligne: https://riptutorial.com/fr/elixir/topic/9519/polymorphisme-chez-elixir

Chapitre 31: Processus

Examples

Créer un processus simple

Dans l'exemple suivant, la `greet` fonction à l' intérieur `Greeter` module est exécuté dans un processus séparé:

```
defmodule Greeter do
    def greet do
        IO.puts "Hello programmer!"
    end
end

iex> spawn(Greeter, :greet, [])
Hello
#PID<0.122.0>
```

Ici, `#PID<0.122.0>` est l' *identificateur de processus* du *processus* généré.

Envoi et réception de messages

```
defmodule Processes do
    def receiver do
        receive do
            {:ok, val} ->
                IO.puts "Received Value: #{val}"
            _ ->
                IO.puts "Received something else"
        end
    end
end
```

```
iex(1)> pid = spawn(Processes, :receiver, [])
#PID<0.84.0>
iex(2)> send pid, {:ok, 10}
Received Value: 10
{:ok, 10}
```

Récursivité et réception

La récursivité peut être utilisée pour recevoir plusieurs messages

```
defmodule Processes do
    def receiver do
        receive do
            {:ok, val} ->
                IO.puts "Received Value: #{val}"
            _ ->
                IO.puts "Received something else"
```

```
        end
        receiver
    end
end
```

```
iex(1)> pid = spawn Processes, :receiver, []
#PID<0.95.0>
iex(2)> send pid, {:ok, 10}
Received Value: 10
{:ok, 10}
iex(3)> send pid, {:ok, 42}
{:ok, 42}
Received Value: 42
iex(4)> send pid, :random
:random
Received something else
```

Elixir utilisera une optimisation de la récursion d'appel tant que l'appel de fonction est la dernière chose qui se produit dans la fonction, comme dans l'exemple.

Lire Processus en ligne: https://riptutorial.com/fr/elixir/topic/3173/processus

Chapitre 32: Programmation fonctionnelle dans Elixir

Introduction

Essayons d'implémenter les fonctions de base des ordres supérieurs comme la carte et de réduire l'utilisation d'Elixir

Examples

Carte

Map est une fonction qui prend un tableau et une fonction et retourne un tableau après avoir appliqué cette fonction à ***chaque élément*** de cette liste.

```
defmodule MyList do
  def map([], _func) do
    []
  end

  def map([head | tail], func) do
    [func.(head) | map(tail, func)]
  end
end
```

Copier coller dans `iex` et exécuter:

```
MyList.map [1,2,3], fn a -> a * 5 end
```

La syntaxe `MyList.map [1,2,3], &(&1 * 5)` est `MyList.map [1,2,3], &(&1 * 5)`

Réduire

Reduce est une fonction qui prend un tableau, une fonction et un accumulateur et utilise l'***accumulateur comme graine pour démarrer l'itération avec le premier élément pour donner le prochain accumulateur et l'itération se poursuit pour tous les éléments du tableau*** (voir l'exemple ci-dessous)

```
defmodule MyList do
  def reduce([], _func, acc) do
    acc
  end

  def reduce([head | tail], func, acc) do
    reduce(tail, func, func.(acc, head))
  end
end
```

Copiez le code ci-dessus dans iex:

1. Pour ajouter tous les nombres dans un tableau: `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`
2. Pour multiplier tous les nombres d'un tableau: `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

Explication par exemple 1:

```
Iteration 1 => acc = 0, element = 1 ==> 0 + 1 ===> 1 = next accumulator
Iteration 2 => acc = 1, element = 2 ==> 1 + 2 ===> 3 = next accumulator
Iteration 3 => acc = 3, element = 3 ==> 3 + 3 ===> 6 = next accumulator
Iteration 4 => acc = 6, element = 4 ==> 6 + 4 ===> 10 = next accumulator = result(as all
elements are done)
```

Filtrer la liste en utilisant réduire

```
MyList.reduce [1,2,3,4], fn acc, element -> if rem(element,2) == 0 do acc else acc ++
[element] end end, []
```

Lire Programmation fonctionnelle dans Elixir en ligne:
https://riptutorial.com/fr/elixir/topic/10186/programmation-fonctionnelle-dans-elixir

Chapitre 33: Programme de base .gitignore pour elixir

Lire Programme de base .gitignore pour elixir en ligne:
https://riptutorial.com/fr/elixir/topic/6493/programme-de-base--gitignore-pour-elixir

Chapitre 34: Programme de base .gitignore pour elixir

Remarques

Notez que le dossier `/rel` n'est peut-être pas nécessaire dans votre fichier .gitignore. Ceci est généré si vous utilisez un outil de gestion des versions tel `exrm`

Examples

Un gitignore de base pour Elixir

```
/_build
/cover
/deps
erl_crash.dump
*.ez

# Common additions for various operating systems:
# MacOS
.DS_Store

# Common additions for various editors:
# JetBrains IDEA, IntelliJ, PyCharm, RubyMine etc.
.idea
```

Exemple

```
### Elixir ###
/_build
/cover
/deps
erl_crash.dump
*.ez

### Erlang ###
.eunit
deps
*.beam
*.plt
ebin
rel/example_project
.concrete/DEV_MODE
.rebar
```

Application d'élixir autonome

```
/_build
/cover
```

```
/deps
erl_crash.dump
*.ez
/rel
```

Application Phoenix

```
/_build
/db
/deps
/*.ez
erl_crash.dump
/node_modules
/priv/static/
/config/prod.secret.exs
/rel
```

.Gitignore généré automatiquement

Par défaut, `mix new <projectname>` générera un fichier `.gitignore` dans la racine du projet qui convient à Elixir.

```
# The directory Mix will write compiled artifacts to.
/_build

# If you run "mix test --cover", coverage assets end up here.
/cover

# The directory Mix downloads your dependencies sources to.
/deps

# Where 3rd-party dependencies like ExDoc output generated docs.
/doc

# If the VM crashes, it generates a dump, let's ignore it too.
erl_crash.dump

# Also ignore archive artifacts (built via "mix archive.build").
*.ez
```

Lire Programme de base .gitignore pour elixir en ligne:
https://riptutorial.com/fr/elixir/topic/6526/programme-de-base--gitignore-pour-elixir

Chapitre 35: Protocoles

Remarques

Une note sur les structs

Au lieu de partager l'implémentation du protocole avec des cartes, les structures nécessitent leur propre implémentation de protocole.

Examples

introduction

Les protocoles permettent le polymorphisme dans Elixir. Définir des protocoles avec `defprotocol` :

```
defprotocol Log do
  def log(value, opts)
end
```

Implémenter un protocole avec `defimpl` :

```
require Logger
# User and Post are custom structs

defimpl Log, for: User do
  def log(user, _opts) do
    Logger.info "User: #{user.name}, #{user.age}"
  end
end

defimpl Log, for: Post do
  def log(user, _opts) do
    Logger.info "Post: #{post.title}, #{post.category}"
  end
end
```

Avec les implémentations ci-dessus, nous pouvons faire:

```
iex> Log.log(%User{name: "Yos", age: 23})
22:53:11.604 [info]  User: Yos, 23
iex> Log.log(%Post{title: "Protocols", category: "Protocols"})
22:53:43.604 [info]  Post: Protocols, Protocols
```

Les protocoles vous permettent d'envoyer à n'importe quel type de données, tant qu'il implémente le protocole. Cela inclut certains types intégrés tels que `Atom` , `BitString` , `Tuples` et autres.

Lire Protocoles en ligne: https://riptutorial.com/fr/elixir/topic/3487/protocoles

Chapitre 36: Sigils

Examples

Construire une liste de chaînes

```
iex> ~w(a b c)
["a", "b", "c"]
```

Construire une liste d'atomes

```
iex> ~w(a b c)a
[:a, :b, :c]
```

Signaux personnalisés

Les signets personnalisés peuvent être créés en créant une méthode `sigil_X` où X est la lettre que vous souhaitez utiliser (il ne peut s'agir que d'une seule lettre).

```
defmodule Sigils do
  def sigil_j(string, options) do
    # Split on the letter p, or do something more useful
    String.split string, "p"
  end
  # Use this sigil in this module, or import it to use it elsewhere
end
```

L'argument `options` est un binaire des arguments donnés à la fin du sigil, par exemple:

```
~j/foople/abc # string is "foople", options are 'abc'
# ["foo", "le"]
```

Lire Sigils en ligne: https://riptutorial.com/fr/elixir/topic/2204/sigils

Chapitre 37: Structures de données

Syntaxe

- [tête | tail] = [1, 2, 3, true] # on peut utiliser la correspondance de motifs pour briser des cellules. Cela assigne la tête à 1 et la queue à [2, 3, vrai]
- % {d: val} =% {d: 1, e: true} # cela assigne val à 1; Aucune variable d n'est créée car le d sur le lhs n'est en réalité qu'un symbole utilisé pour créer le motif% {: d => _} en rubis)

Remarques

En ce qui concerne notre structure de données, voici quelques brèves remarques.

Si vous avez besoin d'une structure de données matricielle, vous devrez écrire beaucoup de listes d'utilisation. Si, au contraire, vous allez lire beaucoup, vous devriez utiliser des tuples.

En ce qui concerne les cartes, elles représentent simplement la manière dont vous stockez les clés.

Examples

Des listes

```
a = [1, 2, 3, true]
```

Notez que ceux-ci sont stockés en mémoire en tant que listes liées. Id est une série de contre-cellules où la tête (List.hd / 1) est la valeur du premier élément de la liste et la queue (List.tail / 1) est la valeur du reste de la liste.

```
List.hd(a) = 1
List.tl(a) = [2, 3, true]
```

Tuples

```
b = {:ok, 1, 2}
```

Les tuples sont l'équivalent des tableaux dans d'autres langues. Ils sont stockés de manière contiguë dans la mémoire.

Lire Structures de données en ligne: https://riptutorial.com/fr/elixir/topic/1607/structures-de-donnees

Chapitre 38: Tâche

Syntaxe

- Task.async (fun)
- Task.await (tâche)

Paramètres

Paramètre	Détails
amusement	La fonction qui doit être exécutée dans un processus séparé.
tâche	La tâche renvoyée par `Task.async` .

Examples

Faire du travail en arrière-plan

```
task = Task.async(fn -> expensive_computation end)
do_something_else
result = Task.await(task)
```

Traitement parallèle

```
crawled_site = ["http://www.google.com", "http://www.stackoverflow.com"]
|> Enum.map(fn site -> Task.async(fn -> crawl(site) end) end)
|> Enum.map(&Task.await/1)
```

Lire Tâche en ligne: https://riptutorial.com/fr/elixir/topic/7588/tache

Chapitre 39: Trucs et astuces

Introduction

Elixir Des astuces et des astuces avancées pour gagner du temps lors du codage.

Examples

Création de Sigils personnalisés et documentation

Chaque x sigil appelle la définition sigil_x respective

Définir des Sigils personnalisés

```
defmodule MySigils do
  #returns the downcasing string if option l is given then returns the list of downcase
letters
  def sigil_l(string,[]), do: String.Casing.downcase(string)
  def sigil_l(string,[?l]), do: String.Casing.downcase(string) |> String.graphemes

  #returns the upcasing string if option l is given then returns the list of downcase letters
  def sigil_u(string,[]), do: String.Casing.upcase(string)
  def sigil_u(string,[?l]), do: String.Casing.upcase(string) |> String.graphemes
end
```

Multiple [OU]

Ceci est juste l'autre façon d'écrire des conditions multiples OU. Ce n'est pas l'approche recommandée car, dans une approche régulière, lorsque la condition est vraie, elle arrête d'exécuter les conditions restantes qui sauvent la durée de l'évaluation, contrairement à cette approche qui évalue d'abord toutes les conditions de la liste. C'est mauvais mais bon pour les découvertes.

```
# Regular Approach
find = fn(x) when x>10 or x<5 or x==7 -> x end

# Our Hack
hell = fn(x) when true in [x>10,x<5,x==7] -> x end
```

Configuration personnalisée iex - Décoration iex

Copiez le contenu dans un fichier et enregistrez le fichier sous le nom de fichier .iex.exs dans votre répertoire de base et consultez la magie. Vous pouvez également télécharger le fichier ICI

```
# IEx.configure colors: [enabled: true]
# IEx.configure colors: [ eval_result: [ :cyan, :bright ] ]
IO.puts IO.ANSI.red_background() <> IO.ANSI.white() <> " ❄❄❄ Good Luck with Elixir ❄❄❄ " <> IO.ANSI.reset
Application.put_env(:elixir, :ansi_enabled, true)
```

```
IEx.configure(
 colors: [
   eval_result: [:green, :bright] ,
   eval_error: [[:red,:bright,"Bug Bug ..!!"]],
   eval_info: [:yellow, :bright ],
 ],
 default_prompt: [
   "\e[G",    # ANSI CHA, move cursor to column 1
    :white,
    "I",
    :red,
    "❤" ,       # plain string
    :green,
    "%prefix",:white,"|",
     :blue,
     "%counter",
     :white,
     "|",
    :red,
    "▶" ,         # plain string
    :white,
    "▶▶"  ,       # plain string
      # ❤ ❤-»" ,  # plain string
    :reset
  ] |> IO.ANSI.format |> IO.chardata_to_string

)
```

Lire Trucs et astuces en ligne: https://riptutorial.com/fr/elixir/topic/10623/trucs-et-astuces

Chapitre 40: Types intégrés

Examples

Nombres

Elixir est livré avec des **nombres entiers** et **des nombres à virgule flottante** . Un **littéral entier** peut être écrit au format décimal, binaire, octal et hexadécimal.

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

Comme Elixir utilise l'arithmétique bignum, **la plage d'entiers n'est limitée que par la mémoire disponible sur le système** .

Les nombres à virgule flottante sont à double précision et respectent la spécification IEEE-754.

```
iex> x = 6.8
6.8

iex> x = 1.23e-11
1.23e-11
```

Notez qu'Elixir prend également en charge la forme exposant pour les flottants.

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

Nous avons d'abord ajouté deux nombres entiers, et le résultat est un entier. Plus tard, nous avons ajouté deux nombres à virgule flottante et le résultat est un nombre à virgule flottante.

La division en Elixir renvoie toujours un nombre à virgule flottante:

```
iex> 10 / 2
5.0
```

De la même manière, si vous ajoutez, soustrayez ou multipliez un nombre entier par un nombre à virgule flottante, le résultat sera en virgule flottante:

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0

iex> 3 * 3.0
9.0
```

Pour la division entière, on peut utiliser la fonction `div/2` :

```
iex> div(10, 2)
5
```

Les atomes

Les atomes sont des constantes qui représentent un nom de quelque chose. La valeur d'un atome est son nom. Un nom d'atome commence par un deux-points.

```
:atom   # that's how we define an atom
```

Le nom d'un atome est unique. Deux atomes avec les mêmes noms sont toujours égaux.

```
iex(1)> a = :atom
:atom

iex(2)> b = :atom
:atom

iex(3)> a == b
true

iex(4)> a === b
true
```

Les booléens `true` et `false` sont en réalité des atomes.

```
iex(1)> true == :true
true

iex(2)> true === :true
true
```

Les atomes sont stockés dans une table d'atomes spéciale. Il est très important de savoir que cette table n'est pas récupérée. Donc, si vous voulez (ou accidentellement c'est un fait) créer constamment des atomes - c'est une mauvaise idée.

Binaires et Bitstrings

Les fichiers binaires dans elixir sont créés à l'aide de la construction Kernel.SpecialForms << >> .

C'est un outil puissant qui rend Elixir très utile pour travailler avec des protocoles et des

encodages binaires.

Les fichiers binaires et les chaînes de bits sont spécifiés à l'aide d'une liste d'entiers ou de valeurs de variable séparés par des virgules, auxquels sont ajoutés "<<" et ">>". Ils sont composés d'unités, soit un groupement de bits, soit un regroupement d'octets. Le regroupement par défaut est un octet unique (8 bits), spécifié à l'aide d'un entier:

```
<<222,173,190, 239>> # 0xDEADBEEF
```

Les chaînes Elixir sont également converties directement en binaires:

```
iex> <<0, "foo">>
<<0, 102, 111, 111>>
```

Vous pouvez ajouter des "spécificateurs" à chaque "segment" d'un binaire, ce qui vous permet de coder:

- Type de données
- Taille
- Endianness

Ces spécificateurs sont encodés en suivant chaque valeur ou variable avec l'opérateur "::":

```
<<102::integer-native>>
<<102::native-integer>> # Same as above
<<102::unsigned-big-integer>>
<<102::unsigned-big-integer-size(8)>>
<<102::unsigned-big-integer-8>> # Same as above
<<102::8-integer-big-unsigned>>
<<-102::signed-little-float-64>> # -102 as a little-endian Float64
<<-102::native-little-float-64>> # -102 as a Float64 for the current machine
```

Les types de données disponibles que vous pouvez utiliser sont les suivants:

- entier
- flotte
- bits (alias pour chaîne de bits)
- chaîne de bits
- binaire
- octets (alias pour binaire)
- utf8
- utf16
- utf32

Sachez que lors de la spécification de la taille du segment binaire, celle-ci varie en fonction du type choisi dans le spécificateur de segment:

- entier (par défaut) 1 bit
- flotteur 1 bit
- binaire 8 bits

Lire Types intégrés en ligne: https://riptutorial.com/fr/elixir/topic/1774/types-integres

Chapitre 41: utilisation de base des clauses de garde

Examples

utilisations de base des clauses de garde

Dans Elixir, on peut créer plusieurs implémentations d'une fonction avec le même nom et spécifier des règles qui seront appliquées aux paramètres de la fonction *avant d'appeler la fonction* afin de déterminer l'implémentation à exécuter.

Ces règles sont marquées par le mot-clé `when` et elles se situent entre le nom de la `def function_name(params)` et le `do` dans la définition de la fonction. Un exemple trivial:

```
defmodule Math do

  def is_even(num) when num === 1 do
    false
  end
  def is_even(num) when num === 2 do
    true
  end

  def is_odd(num) when num === 1 do
    true
  end
  def is_odd(num) when num === 2 do
    false
  end

end
```

Disons que je lance `Math.is_even(2)` avec cet exemple. Il existe deux implémentations de `is_even` , avec des clauses de garde différentes. Le système les examinera dans l'ordre et exécutera la première implémentation où les paramètres satisfont à la clause de protection. Le premier spécifie que `num === 1` qui n'est pas vrai, donc il passe au suivant. Le second spécifie que `num === 2` , ce qui est vrai, c'est donc l'implémentation utilisée et la valeur de retour sera `true` .

Et si je lance `Math.is_odd(1)` ? Le système examine la première implémentation et constate que, puisque `num` est `1` la clause guard de la première implémentation est satisfaite. Il utilisera alors cette implémentation et renverra `true` , sans avoir à chercher d'autres implémentations.

Les gardes sont limités dans les types d'opérations qu'ils peuvent exécuter. La documentation Elixir répertorie toutes les opérations autorisées . en bref, ils permettent des comparaisons, des opérations mathématiques, des opérations binaires, une vérification de type (par exemple, `is_atom`) et une poignée de petites fonctions pratiques (par exemple, la `length`). Il est possible de définir des clauses de garde personnalisées, mais cela nécessite la création de macros et il est préférable de laisser un guide plus avancé.

Notez que les gardes ne jettent pas d'erreurs; ils sont traités comme des défaillances normales de la clause de protection et le système passe à la prochaine implémentation. Si vous trouvez `(FunctionClauseError) no function clause matching` lors de l'appel d'une fonction protégée avec des paramètres que vous prévoyez de travailler, il se peut qu'une clause de protection que vous prévoyez utiliser génère une erreur qui est en train d'être avalée.

Pour voir cela par vous-même, créez puis appelez une fonction avec une garde qui n'a aucun sens, comme celle qui essaie de diviser par zéro:

```
defmodule BadMath do
  def divide(a) when a / 0 === :foo do
    :bar
  end
end
```

L'appel de `BadMath.divide("anything")` fournira l'erreur peu utile `(FunctionClauseError) no function clause matching in BadMath.divide/1` - alors que si vous aviez essayé de lancer `"anything" / 0` directement, vous obtiendriez une aide plus utile error: `(ArithmeticError) bad argument in arithmetic expression`.

Lire utilisation de base des clauses de garde en ligne:
https://riptutorial.com/fr/elixir/topic/6121/utilisation-de-base-des-clauses-de-garde

Crédits

S. No	Chapitres	Contributeurs
1	Premiers pas avec Elixir Language	alejosocorro, Andrey Chernykh, Ben Bals, Community, cwc, Delameko, Douglas Correa, helcim, I Am Batman, JAlberto, koolkat, leifg, MattW., rap-2-h, Simone Carletti, Stephan Rodemeier, Vinicius Quaiato, Yedhu Krishnan, Zimm i48
2	Cartes et listes de mots clés	Sam Mercier, Simone Carletti, Yos Riady
3	Comportements	Yos Riady
4	Conditionnels	Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
5	Conseils de débogage	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU, sbs
6	Conseils et astuces pour la console IEx	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
7	Cordes	Alex G, Sheharyar, Yos Riady
8	Correspondance de motif	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile , Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan
9	Courant	Oskar
10	Des listes	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
11	Doctests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
12	Ecto	fgutierr, Philippe-Arnaud de MANGOU, toraritte
13	Erlang	4444, Yos Riady
14	ExDoc	milmazz, Yos Riady
15	ExUnit	Yos Riady
16	FAISCEAU	Yos Riady

17	Installation	cwc, Douglas Correa, Eiji, JAlberto, MattW.
18	Joindre des chaînes	Agung Santoso
19	Les constantes	ibgib
20	Les fonctions	Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan , Yos Riady
21	Les nœuds	Yos Riady
22	Les opérateurs	alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
23	Manipulation d'Etat à Elixir	Paweł Obrok
24	Meilleur débogage avec IO.inspect et les étiquettes	leifg
25	Mélanger	4444, helcim, rainteller, Slava.K, Yos Riady
26	Métaprogrammation	4444, Paweł Obrok
27	Modules	Alex G, javanut13, Yos Riady
28	Obtenir de l'aide dans la console IEx	helcim
29	Optimisation	Filip Haglund, legoscia
30	Polymorphisme chez Elixir	mustafaturan
31	Processus	Alex G, Yedhu Krishnan
32	Programmation fonctionnelle dans Elixir	Dinesh Balasubramanian
33	Programme de base .gitignore pour elixir	Yos Riady
34	Protocoles	Yos Riady
35	Sigils	javanut13, Yos Riady
36	Structures de données	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady

37	Tâche	mario
38	Trucs et astuces	Ankanna
39	Types intégrés	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
40	utilisation de base des clauses de garde	alxndr